

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

Broadview
www.broadview.com.cn

精通实时大数据分析

Druid

实时大数据分析

原理与实践

欧阳辰 刘麒赞 张海雷 高振源 等著

腾讯、小米、优酷、云测等互联网公司的一线实践经验
为你解读海量实时OLAP平台



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn



欧阳辰

小米商业产品部研发总监，负责广告架构和数据分析平台，擅长数据挖掘，大数据分析和广告搜索架构。之前，在微软工作10年，任微软公司高级开发经理，负责Contextual Ads产品研发，开发Bing Index Serve的核心模块。持有多项关于互联网广告及搜索的美国专利。创办“互联居”公众号，致力于互联网广告技术的繁荣。毕业于北京大学计算机系，获硕士学历。



刘麒雯

现任Testin云测公司技术总监，全面负责领导团队完成数据分析产品的研发。作为资深数据技术专家，曾为多个著名开源项目（Hadoop / Sqoop / Oozie / Druid）贡献源代码，在互联网大数据分析、机器学习和统计学应用等方面拥有丰富的实战经验和相关专利。在企业级产品研发和客户支持方面也有着丰富的经验，并曾为中国多地（包括香港和台湾地区）的龙头企业成功进行实地支持，为美国与新加坡等地客户进行远程支持。之前，曾任OneAPM公司大数据架构师，以及在IBM公司工作七年并任IBM全球大数据平台产品BigInsights的Advisory Software Engineer。

Druid

实时大数据分析

原理与实践

欧阳辰 刘麒赞 张海雷 高振源 许哲 著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

Druid 作为一款开源的实时大数据分析软件,最近几年快速风靡全球互联网公司,特别是对于海量数据和实时性要求高的场景,包括广告数据分析、用户行为分析、数据统计分析、运维监控分析等,在腾讯、阿里、优酷、小米等公司都有大量成功应用的案例。本书的目的就是帮助技术人员更好地深入理解 Druid 技术、大数据分析技术选型、Druid 的安装和使用、高级特性的使用,也包括一些源代码的解析,以及一些常见问题的快速回答。

Druid 的生态系统正在不断扩大和成熟,Druid 也正在解决越来越多的业务场景。希望本书能帮助技术人员做出更好的技术选型,深度了解 Druid 的功能和原理,更好地解决大数据分析问题。本书适合大数据分析的从业人员、IT 人员、互联网从业者阅读。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

Druid 实时大数据分析原理与实践 / 欧阳辰等著. —北京: 电子工业出版社, 2017.3

ISBN 978-7-121-30623-5

I. ① D…II. ① 欧…III. ① 数据处理 IV. ① TP274

中国版本图书馆 CIP 数据核字(2016)第 304239 号

策划编辑: 符隆美

责任编辑: 葛 娜

印 刷: 三河市良远印务有限公司

装 订: 三河市良远印务有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787×980 1/16 印张: 22

字数: 478 千字

版 次: 2017 年 3 月第 1 版

印 次: 2017 年 3 月第 1 次印刷

印 数: 4000 册 定价: 79.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式: (010) 51260888-819 faq@phei.com.cn。

Foreword

Like many popular open source projects, Druid was initially created to solve a problem. We were trying to build an interactive analytics UI at a small advertising technology startup in San Francisco, and struggled to find a technology that could rapidly aggregate, slice and dice, and drill down into massive data sets. Eric Tschetter started the first lines of Druid to tackle this challenge, and that work has somehow led to an international community forming around the project.

I joined Eric on Druid soon after the project started, and for a while, the Druid world consisted of only 2 engineers. The first version of Druid was extremely minimalistic; there was a single process type, the “compute” node, and a handful of queries, but the core that was there was just enough to solve the problems with scale and performance we had at that time.

Our Druid cluster in the early years was less than 20 nodes, and we worked around the clock to aggressively develop features and fix bugs. There were a lot of late nights in those days. I can still very clearly recall waking up in the middle of the night to fix an outage, and occasionally cursing loudly because the only reason the pager went off was because it was out of batteries.

As Druid matured, and as data volumes grew, we continued to face challenges around performance at scale and operational stability. Running in the then notoriously finicky Amazon Web Services cloud environment wasn’t always easy, and led us to make the decision to break up “compute” nodes into different components so that individual components could be fine tuned at scale, and any one component could fail without impacting the functionality of the other components. I am glad we made those decisions because it led us to sleep much more at night.

It has been extremely rewarding to watch the grassroots growth of the open source community. Unlike other popular open source projects, Druid was not developed at a major technology company or famous research lab. We open sourced the project without much attention, and the first open source version of the project almost didn’t have querying capabilities. We weren’t allowed to open source

many pieces of the codebase, including most of the queries we developed. The night before officially announced the project, Eric was up writing GroupBy queries in a hotel room just so people could have a way of getting data out of Druid. After we released Druid, the code repository was completely undocumented and barely functional. I don't think a single organization tried to use Druid when it was first open sourced.

I've long lost count of how many companies actually run Druid in production today, but I am glad people have found value from our work. I was very excited to learn that Qiyun Liu was writing a book on Druid. I hope through his book, you will learn much more about our project, and learn how to leverage it to bring value to your organization.

Fangjin Yang

Co-Founder, Druid

Co-Founder and CEO, Imply

San Francisco, California

2016.11.20

序言

正如许多广为应用的开源项目，Druid 是为解决某个特定问题而诞生的。几年前，在旧金山的一家广告技术创业公司，我们想要创建一个交互式分析的用户界面，同时也在寻找可以快速聚合、切片并深挖海量数据集的技术。为了解决这个技术难题，Eric Tschetter 开始了 Druid 项目的第一行代码。到目前为止，这个项目已经拥有了跨越多个国家的社区，并且在不断发展壮大。

在 Eric 开始了 Druid 项目后，我很快便加入了这个团队。说是团队，实际上很长一段时期只有我和 Eric 两个人。Druid 的最初版本极其简单：只有一种分析类型、几个“计算”节点，以及一些简陋的查询功能。但是 Druid 的这些核心功能却已经足以解决我们当时所面对的性能与规模的难题。

在早期时，Druid 集群只有不到 20 个节点。那时我们会日以继夜地工作，不断地开发新的功能与改正代码中的错误。在那些日子里，我们经常会工作到深夜，直到现在我还能清楚地记得自己在半夜爬起来修复瘫痪的系统。不过有时半夜被叫醒的原因也会让我们感到暗暗生气，因为系统告警的原因竟然是由于呼叫器的电池没电了。

随着 Druid 的逐渐成熟，以及数据量的持续增长，我们在集群规模和运营稳定性方面不断面临新的挑战。众所周知，当时亚马逊网络服务云环境的状况不是很好，甚至有点儿“臭名昭著”，因此在它上面运行集群其实并不容易，从而促使我们决定将“计算”节点分解成不同的独立组件，以便可以在大规模的集群上对单个组件进行微调，同时保证任何一个组件的失败不会影响到其他组件的正常运行。现在回头看，我很高兴我们当初做了这些决定。正因为有了这些高容错的功能，我们终于可以不用时不时半夜起床，从而可以多睡一会儿了。

亲眼见证了这个项目的成长过程让我非常欣慰。与其他大型的开源项目不同，Druid 不是在一家前沿技术公司或是享誉盛名的研究实验室开发的。这个项目刚开源的时候并没有受到很多关注，而且第一个开源版本甚至几乎没有查询功能。许多代码库，包括我们当时开发的大多数查询功能，由于没有得到当时公司的允许从而没能开源。在正式宣布这个项目的

很显然,因为行业的多样性,业务场景变得越来越复杂,对数据处理的要求已经不仅是体量大和速度快,还要数据结构灵活、编程接口强大、系统可扩展、原子化操作、高效备份、读性能加速或者写性能加速等。在这个技术普及的时代,不仅互联网行业有越来越多的技术人员和数据人员开始参与到大数据工作中,而且很多传统软件从业者也慢慢受到吸引,双方互相借鉴,进一步扩大了大数据技术的能力和影响。可以看到,传统的数据库、操作系统、编程语言等技术思想被引入来解决各种复杂的需求。因此而诞生的包括 NoSQL、SQL on Hadoop、ElasticSearch 这样的新事物,逐渐把我们推进到一个全新的时代。

3. 创新时代

本书所介绍的 Druid,是大数据技术新时代的产物。现在的新技术,并不只是解决各种技术问题,而是更加贴近复杂的创新型业务的需求场景。我们看到,业内的新框架和新产品,都在探索如何让大数据能为各种不同类型的业务带来更多的优化,解决数据可用性、垂直性、实时性、灵活性、可视化等问题。

如本书所介绍的,Druid 以及相关配套的工作,使我们可以非常灵活地实时分析数据,做复杂的维度切割和条件查询,而且可以非常方便地做可视化展示。无论是在互联网企业,还是传统企业,这个工具的使用场景都是非常丰富的,如监控报警、诊断排错、生成业务报表、对接机器学习及策略优化等。

在这个创新时代,还有很多新技术涌现出来,比如强调可编程与实时性的 Spark、与 Druid 类似的 Pinot,还有 A/B 测试(比如我们吆喝科技提供的解决方案)等。

可以看到,大数据相关技术的发展速度是逐渐加快的。原因自然是相关应用的普及(本书有很多详尽的相关案例介绍多家成功公司的应用场景),以及因此而带来的从业人员规模的增长(感谢互联网行业招募和培养了大批人才)。

从 MapReduce 论文 2004 年问世到 Apache Hadoop 框架被广泛使用,经过了 5 年以上的时间。而从 Dremel 论文问世到 Druid 被广泛认可,只用了 3 年时间。值得指出的是,在这几年时间内,还有很多公司借鉴了 Lambda Calculus 思想自己研发了闭源系统(Microsoft Dryad、阿里巴巴等)。不过经过几年的实践摸索,业内逐渐形成了以 Apache 的一系列项目为核心的统一解决方案。大家逐渐意识到,与其对同一问题采取不同的解决方案,不如一个问题一个解决方案,然后大家一起来探索解决更多不同场景的问题。这是现代互联网时代特有的网络效应和规模效应。

本书很大的贡献就是普及 Druid(以及如 Pinot 这样的相似框架),让更多的技术人员、数据人员和互联网业务人员可以快速地熟悉和尝试这个成功的新技术,将它应用在更多场景中,然后能激发更多的创新,进一步推动 Druid 以及新技术的持续发展。

我在 Google 总部工作的时候，经常使用 Dremel（和 Druid 类似的工具），也用过基于 Dremel 的可视化系统 PowerDrill。当时的感觉是，一个像 SQL 一样好用的工具，却能快速查询海量的实时的数据，对业务帮助非常大。举个例子，当时某个广告新产品上线测试后数据不佳，Dremel 从实时的数据里发现在某些浏览器里没有点击，于是进一步发现在这些浏览器里渲染有问题，马上改正。如果没有 Dremel，这个问题的解决可能需要至少 1 周以上，而不是几个小时。相信开源的 Druid 也会像 Dremel 一样，在很多企业内成为业务数据分析的利器，大幅度提高大家的工作效率。

本书特别出色的地方在于，不仅对 Druid 的架构以及细节有深入的阐述，而且有非常详尽的代码例子（codelab），甚至有一章专门介绍怎么安装和配置，非常适合工程师一边学习，一边上机实践。在 Druid 项目文档还不是特别完善的情况下，这本书不仅适合作为大家的学习材料，还能当作日常工作中的手册，已备随时查询。

本书的作者欧阳辰是大数据领域的顶级专家，他现在服务的小米公司在大数据创新上非常积极，对于 Druid 的使用和贡献也处于业内领先的地位。所以，本书里有非常多的真实业务场景相关的解析，不仅对技术人员，而且对数据人员和业务人员也非常有借鉴价值。

如果你想拥抱大数据的新时代，Druid 是你的必学，本书是你的必读。

王晔

AdHoc 吆喝科技 创始人 CEO

推荐序二

向在大数据行业从事多年的架构师、正在如火如荼地开展大数据相关工作的工程师，以及正在准备步入大数据行业的新手推荐《Druid 实时大数据分析原理与实践》这本书。

在从北京到苏州的高铁上花了 5 个小时读了这本书，虽然还没有读完，但是我已经可以非常确定地告诉大家这本书“非常引人入胜”。对我这样一个在软件行业做了 30 年的“码农”、15 年以上互联网从业的老兵来说，也别开生面地学了很多新知识，又把脑中的和大数据有关的各种系统知识重新更新了一遍。

本书非常清晰、明确地介绍了 Druid 是什么、是为什么设计的、特点和特长是什么，以及如何使用。

本书在介绍美国 MetaMarkets 公司为什么会设计 Druid 的同时介绍了业界流行的和大数据有关的大部分系统，以及这些系统诞生的原因及相互之间的比较和特长，比如经典的 Hadoop、飞速发展的 Spark、用于实时数据流的 kafka，非常引人入胜。

本书在介绍为什么和如何使用 Druid 的同时介绍了 Druid 的源代码结构，对那些心里痒痒地想给 Druid 做点贡献的工程师开启了一条入门的道路。

本书最后一章“Druid 生态与展望”很好地介绍了在先行使用 Druid 的用户中逐渐开发的配套设施，以及这些配套设施如何反过来帮助 Druid 的发展。想使用或者评估 Druid 的用户都能从这一章得到很多新的启示，并节省用来评估和寻找 Druid 相关配套设施的时间。

Sherman Tong

微软中国研发中心，高级研发总监

推荐语

(排名不分先后)

只有久经考验又乐于分享的大数据架构师，才有这样的功力，把实时大数据分析技术的原理与实践讲得这么系统与透彻。书中随处可见来自实践的真知灼见。阅读这本书，就如同由一位老司机带着开启的美妙旅程，一路轻松、兴奋、风景无限。

鲁肃

蚂蚁金服 CTO

无论是数据量总量还是数据增量都在急速增长的背景下，急需一种技术能够快速地对海量数据进行实时存储和多维度实时分析，Druid 作为一款优秀的实时大数据分析引擎应运而生。Druid 非常强大，与之伴随的是使用上的复杂性，因此理解 Druid 的架构和运行机制原理对于更好地使用 Druid 及定制化扩展显得尤为重要。《Druid 实时大数据分析原理与实践》这本书正好可以满足读者的需求。诸位作者理论功底深厚，实践经验丰富，本书可以帮助大家快速地了解和学习 Druid，强烈推荐。

张雪峰

饿了么 CTO

开源软件已经成为了构建现代软件系统的重要基石，特别是在大数据和云计算等热门领域，开源软件更是独领风骚。作为一家技术驱动的公司，Testin 云测一直是开源软件坚定的倡导者和实践者，在 Testin 各个产品线中都使用了开源技术，Testin 云测是开源的受益者。其中，大数据实时多维度分析场景充满技术挑战，很高兴看到 Druid 最终完美地解决了我们客户的问题。大数据时代已经到来，Druid 无疑是解决大数据多维度实时分析的最佳选择，本书则是一把打开该技术之门的钥匙。

徐琨

Testin 云测总裁

2015 年, 我们因大数据实时分析的业务需求而开始接触 Druid。在做架构选型时, Druid 因其在快速查询、水平扩展、实时数据摄入和分析这三方面都有良好的支持而很好地满足了我们的需求。2016 年年初, 我们几个 Druid 技术爱好者和 Druid 联合创始人 Fangjin Yang 一起组建了 Druid 中国用户组的微信群, 并举办了多次 Druid Meetup。靠着技术圈同学的口口相传, Druid 中国用户组从最初十几个人的小群, 已发展为 500 人的大群; 与此同时, 阿里、腾讯、小米、滴滴等众多公司也都开始使用 Druid。本书的几位作者都是 Druid 中国用户组中非常活跃的技术专家, 他们在社区中的口碑是本书质量的保证, 如果你对 Druid 感兴趣, 这本书一定不能错过。

陈冠诚

Druid 中国用户组发起人

从 2011 年创业开始, TalkingData 就是开源技术社区的重度参与者, 因为我们始终面临海量数据的压力, 仅靠自己闭门造车完全行不通。我们自 2013 年开始关注 Druid 项目, 因为它的特性非常契合分析的业务场景, 能解决海量数据的多维交叉分析问题。同时, 为了增强其分析能力, 我们也在把基于 Bitmap 的自研分析引擎 Atom Cube 融合到 Druid 中。拥抱开源社区的各种曲折, 有苦有乐, 不足道也, 但是庆幸有许多热情的领路人, 给大家无私的帮助。本书作者之一欧阳辰就是这样一位乐于分享的人, 文理兼修, 对技术和数据都有深厚的积累和独到的见解, 让人敬佩。相信这本书一定能够带大家领略 Druid 的魅力, 让大家少走弯路, 真正聚焦在对数据的探索上。

肖文峰

TalkingData CTO

Druid 正在开创海量数据实时数据分析的时代, 作为一家以技术创新驱动的公司, OneAPM 幸运地在正确的时间选择了正确的技术构筑自己的后端处理平台, 我希望 OneAPM 的经验能够给后来者以借鉴, 本书作者之一麒麟是 Druid 技术在 OneAPM 落地生根的实践者, 这本书一定能够给大家更多的启迪。

何晓阳

OneAPM 创始人, 董事长

开源软件在过去十年中蓬勃发展, 特别是在大数据等新兴领域, 开源软件逐渐在企业级应用中占有一席之地。我们很欣喜地看到 Druid 这样有中国元素的开源项目在这个过程中茁壮成长, 被企业客户接受并在核心系统应用中部署。

刘隶放

Cloudera 大中华区技术总监

我用“大、全、细、时”四个字来总结大数据，传统数据库在这种数据特性下根本无法支撑。而 Druid 的出现，正好比较完美地满足了这四点，特别是对于维度变换不频繁的场景，非常适用。本书既讲解了 Druid 技术本身，也讲解了多维数据分析相关的知识，并对业内的分布式存储和查询系统都做了对比。想要系统掌握 Druid 技术，推荐阅读本书。

桑文锋

神策数据公司创始人 & CEO

Druid 是一个分布式的支持实时分析的数据存储系统 (DataStore)，Druid 设计之初就是为分析而生的，主要应用于大数据实时查询和分析的高容错、高性能开源分布式系统，旨在快速处理大规模的数据，并能够实现快速查询和分析。本书让读者能够深入了解 Druid 的架构设计、设计理念、安装配置、集群管理和监控，书中还介绍了一些高级特性和核心源码的导读，最后深入分析了 Druid 的最佳实践。本书采用由浅入深、循序渐进的方式介绍 Druid，是一本非常难得的 OLAP 的实时分析系统经典书籍。

卢亿雷

AdMaster (精硕科技) 技术副总裁

前言

大数据的繁荣已经来到，Druid 是数据分析的一把利剑，以开源之道，高效解决了大数据实时分析的众多场景！希望此书成为 Druid 宝剑秘籍，帮助读者利用 Druid 解决业务问题。

大数据，相比传统数据，具有四个典型特征，即形式多、体量大、速度快及价值高，最终以产生商业和社会价值为目的。Druid 在解决大数据问题时，能够比较好地处理其中两个方面，一是大数据量；二是实时处理速度。Druid 在设计上支持 PB 级别的数据处理能力，在实践中，不少公司都有几百 TB 数据量的成功应用。实时性是 Druid 的一个内置特性，能够轻松应对每秒数万的流式实时事件，并且支持水平扩展。Druid 的大多数数据查询的响应时间也在亚秒级。在数据多样性方面，Druid 还是有些局限性的，它只支持强类型的数据结构，原因是为了保证数据索引的高效性和查询性能。

2005 年以前，数据分析主要是以关系型数据库为基础，包括多维数据库（OLAP），支持中小规模数据的复杂维度的分析查询。2005 年以后，很多分析场景的数据量大大增加，对实时性要求高，对计算总成本敏感，主流关系型数据库开始有些力不从心。同时，数据行业涌现出一批基于新硬件架构设计的数据存储系统，常常统称为 NoSQL，很多系统都高效地解决了 Key-Value 的存储和访问问题。

2010 年后，基于大数据的分析场景越来越多、越来越重要。大数据驱动的业务模式开始深入人心，没有数据指标，就无法进行优化。很多数据软件系统都无法支持 TB 到 PB 级别数据量的实时分析，Druid 就是在这种背景下脱颖而出的开源软件，帮助大家解决业务中的实时数据分析问题。

2011 年，MetaMarkets 公司为了解决广告交易中海量实时数据的分析问题，在尝试了各种 SQL 和 NoSQL 方案后，最后决定自行设计且创建了 Druid，该项目于 2013 年开源。Druid 是一个支持在大型数据集上进行实时查询而设计的开源数据分析和存储系统，提供了低成本、高性能、高可靠性的解决方案，整个系统支持水平扩展，管理方便。实际上，Druid 的很多设计思想来源于 Google 的秘密分析武器 PowerDrill，在功能上，和 Apache 开源的 Drill 也有几分相似。Druid 被设计成支持 PB 级别的数据量，现实中有数百 TB 级别的数据应用实

例，每天处理数十亿流式事件。Druid 之所以保持高效，有这样几个原因：一是数据进行了有效的聚合或预计算；二是数据结构的优化，应用了 Bitmap 的压缩算法；三是可扩展的高可用架构，灵活支持部署的扩展；四是社区的力量，Druid 开发和用户社区保持活跃，不断推动 Druid 的完善和改进。

Druid 成功应用于众多互联网和非互联网公司中，特别是用户行为分析、个性化推荐的数据分析、物联网的实时数据分析、互联网广告交易分析等领域。国内的主流广告技术公司，都曾尝试或开始采用 Druid 支持实时数据分析。传统技术公司如 Cisco, SK Telecom，也都在使用 Druid 进行用户行为分析等项目。Druid 帮助这些业务场景实现了高效数据存储和流式数据分析。

另外，Druid 项目中也有不少中国元素，其创始人之一为中国工程师杨仿今，其他核心开发工程师也包括阿里的宾莉金、谷歌的郭秉坤等。杨仿今曾多次来到中国进行 Druid 的技术交流。Druid 项目初期，不少中国广告技术公司参与了 Druid 的技术评估。目前该技术也广泛应用于中国互联网公司中，例如腾讯、阿里、小米、优酷土豆、蓝海讯通等。

本书的目的就是介绍 Druid，让读者能够深入了解 Druid 的架构设计、使用管理，也介绍了一些高级特性和核心源码的导读。本书采用由浅入深、循序渐进的方式介绍 Druid，内容组织如下：

第 1 章，介绍 Druid 的初级概念；第 2 章，对行业中不同的数据分析软件进行介绍和对比，包括一些时序数据库；第 3 章，Druid 的设计理念和架构介绍；第 4 章，Druid 的安装和配置；第 5 章，Druid 的数据摄入；第 6 章，查询详解；第 7 章，介绍 Druid 的一些高级特性，包括正在积极完善的一些功能；第 8 章，核心代码的导读和分析；第 9 章，集群管理中的安全和监控；第 10 章，介绍几个公司的 Druid 最佳实践；第 11 章，Druid 的生态介绍和展望。附录 A 简要回答了一些常见的问题；附录 B 列出了各个服务模块的参数含义和建议值，方便系统管理。

Druid 本身也在不断升级中，大约每 3~6 个月都有一次升级，每年都有一个大变化，支持更多的业务场景，不断支持各种主流开源生态，同时围绕着 Druid 的开源生态也在慢慢崛起，包括灵活数据摄入、数据可视化、标准 SQL 查询等。目前 Druid 在中国发展非常迅速，几乎都是口口相传的推广，很多基于 Druid 的项目都是线上产品的一部分。正因如此，我们有理由相信 Druid 必将会在开源世界里拥有一个更为繁荣和光彩的明天！

本书创作组

目录

第1章 初识 Druid	1
1.1 Druid 是什么	1
1.2 大数据分析和 Druid	1
1.3 Druid 的产生	3
1.3.1 MetaMarkets 简介	3
1.3.2 失败总结	4
1.4 Druid 的三个设计原则	4
1.4.1 快速查询 (Fast Query)	5
1.4.2 水平扩展能力 (Horizontal Scalability)	5
1.4.3 实时分析 (Realtime Analytics)	6
1.5 Druid 的技术特点	6
1.5.1 数据吞吐量	6
1.5.2 支持流式数据摄入	6
1.5.3 查询灵活且快	6
1.5.4 社区支持力度大	7
1.6 Druid 的 Hello World	7
1.6.1 Druid 的部署环境	7
1.6.2 Druid 的基本概念	7
1.7 系统的扩展性	9
1.8 性能指标	10
1.9 Druid 的应用场景	10
1.9.1 国内公司	11

1.9.2 国外公司	12
1.10 小结	13
参考资料	13
第2章 数据分析及相关软件	15
2.1 数据分析及相关概念	15
2.2 数据分析软件的发展	16
2.3 数据分析软件的分类	17
2.3.1 商业软件	17
2.3.2 时序数据库	22
2.3.3 开源分布式计算平台	23
2.3.4 开源分析数据库	25
2.3.5 SQL on Hadoop/Spark	31
2.3.6 数据分析云服务	33
2.4 小结	34
参考资料	34
第3章 架构详解	35
3.1 Druid 架构概览	35
3.2 Druid 架构设计思想	36
3.2.1 索引对树结构的选择	37
3.2.2 Druid 总体架构	41
3.2.3 基于 DataSource 与 Segment 的数据结构	43
3.3 扩展系统	45
3.3.1 主要的扩展	45
3.3.2 下载与加载扩展	46
3.4 实时节点	47
3.4.1 Segment 数据文件的制造与传播	47
3.4.2 高可用性与可扩展性	48
3.5 历史节点	49
3.5.1 内存为王的查询之道	49

3.5.2	层的分组功能	50
3.5.3	高可用性与可扩展性	51
3.6	查询节点	51
3.6.1	查询中枢点	51
3.6.2	缓存的使用	52
3.6.3	高可用性	52
3.7	协调整点	53
3.7.1	集群数据负载均衡的主宰	53
3.7.2	利用规则管理数据生命周期	53
3.7.3	副本实现 Segment 的高可用性	54
3.7.4	高可用性	54
3.8	索引服务	54
3.8.1	主从结构的架构	54
3.8.2	统治节点	55
3.8.3	中间管理者与苦工	56
3.8.4	任务	56
3.9	小结	57
第4章 安装与配置		58
4.1	安装准备	58
4.1.1	安装包简介	58
4.1.2	安装环境	59
4.1.3	Druid 外部依赖	60
4.2	简单示例	61
4.2.1	服务运行	61
4.2.2	数据导入与查询	62
4.3	规划与部署	65
4.4	基本配置	68
4.4.1	基础依赖配置	68
4.4.2	数据节点配置调优	69
4.4.3	查询节点配置调优	69

4.5	集群节点配置示例	70
4.5.1	节点规划	70
4.5.2	Master 机器配置	72
4.5.3	Data 机器配置	76
4.6	小结	79
第5章	数据摄入	80
5.1	数据摄入的两种方式	80
5.1.1	流式数据源	80
5.1.2	静态数据源	81
5.2	流式数据摄取	81
5.2.1	以 Pull 方式摄取	82
5.2.2	用户行为数据摄取案例	86
5.2.3	以 Push 方式摄取	89
5.2.4	索引服务任务相关管理接口	91
5.3	静态数据批量摄取	94
5.3.1	以索引服务方式摄取	94
5.3.2	以 Hadoop 方式摄取	96
5.4	流式与批量数据摄取的结合	99
5.4.1	Lambda 架构	99
5.4.2	解决时间窗口问题	100
5.5	数据摄取的其他重要知识	101
5.5.1	数据分片	101
5.5.2	数据复制	106
5.5.3	索引服务之 Tranquility	107
5.5.4	高基数维度优化	111
5.6	小结	116
第6章	数据查询	117
6.1	查询过程	117

6.2	组件	118
6.2.1	Filter	118
6.2.2	Aggregator	121
6.2.3	Post-Aggregator	125
6.2.4	Search Query	129
6.2.5	Interval	129
6.2.6	Context	130
6.3	案例介绍	131
6.4	Timeseries	134
6.5	TopN	138
6.6	GroupBy	144
6.7	Select	149
6.8	Search	151
6.9	元数据查询	153
6.10	小结	156
第7章	高级功能和特性	157
7.1	近似直方图 (Approximate Histogram)	158
7.1.1	分位数和直方图	158
7.1.2	实现原理	158
7.1.3	如何使用	161
7.1.4	近似直方图小结	163
7.2	数据 Sketch	163
7.2.1	DataSketch Aggregator	163
7.2.2	DataSketch Post-Aggregator	167
7.3	地理查询 (Geographic Query)	170
7.3.1	基本原理	170
7.3.2	空间索引 (Spatial Indexing)	171
7.3.3	空间过滤 (Spatial Filter)	171
7.3.4	边界条件 (Boundary Condition)	172
7.3.5	地理查询小结	172

7.4 Router	172
7.4.1 Router 概览	172
7.4.2 路由规则	174
7.4.3 配置	175
7.4.4 路由策略	175
7.5 Kafka 索引服务	177
7.5.1 设计背景	177
7.5.2 实现	178
7.5.3 如何使用	182
7.6 Supervisor API	186
7.6.1 创建 Supervisor	186
7.6.2 关闭 Supervisor	186
7.6.3 获取当前执行的 Supervisor	186
7.6.4 获取 Supervisor 规范	186
7.6.5 获取 Supervisor 的状态报告	186
7.6.6 获取所有 Supervisor 的历史	187
7.6.7 获取 Supervisor 的历史	187
7.7 最佳实践	187
7.7.1 容量规划	187
7.7.2 Supervisor 的持久化	187
7.7.3 Schema 的配置与变更	188
7.8 小结	188
第8章 核心源代码探析	189
8.1 如何编译 Druid 代码	189
8.2 Druid 项目介绍	190
8.3 索引结构模块和层次关系	192
8.4 Column 结构	192
8.5 Segment	195
8.6 Query 模块	203
8.6.1 基础组件	203

8.6.2	内存池管理	206
8.6.3	查询流程概览	207
8.6.4	查询引擎	225
8.7	Coordinator 模块	229
8.8	小结	237
第 9 章	监控和安全	238
9.1	Druid 监控	238
9.1.1	Druid 监控指标	238
9.1.2	常用的监控方法	245
9.2	Druid 告警	250
9.2.1	Druid 告警信息	250
9.2.2	Druid 与告警系统的集成	250
9.3	Druid 安全	251
9.3.1	Druid 与利用 Kerberos 加强安全认证的系统集成	251
9.3.2	集成外部权限模块完成用户授权	255
9.4	小结	256
第 10 章	实践和应用	257
10.1	小米	257
10.1.1	场景一：小米统计服务	258
10.1.2	场景二：广告平台实时数据分析	260
10.2	优酷土豆	262
10.2.1	需求分析	262
10.2.2	技术选型及工程实践	263
10.2.3	优化策略	266
10.3	腾讯	267
10.3.1	工程实践	267
10.3.2	业务实践	270
10.4	蓝海讯通	279
10.5	小结	284

第 11 章 Druid 生态与展望	285
11.1 Druid 生态系统	285
11.2 Druid 生态系统资源	288
11.2.1 IAP	288
11.2.2 Plywood	289
11.2.3 PlyQL	294
11.2.4 Pivot	297
11.2.5 Druid-Metrics-Kafka	300
11.2.6 Caravel (Airbnb)	301
11.3 Druid 的社区讨论组	302
11.4 Druid 展望	302
参考资料	303

附录 A 常见问题 (FAQ)	304
------------------------------	------------

附录 B 常用参数表	312
-------------------------	------------

第1章

初识 Druid

1.1 Druid 是什么

Druid 单词来源于西方古罗马的神话人物，中文常常翻译成德鲁伊。传说 Druid 教士精通占卜，对祭祀礼仪一丝不苟，也擅长于天文历法、医药、天文和文学等。同时，Druid 也是执法者、吟游诗人和探险家的代名词。在著名游戏《暗黑破坏神》中，Druid 也是一个非常有战斗力的人物，法术出色。

本书介绍的 Druid 是一个分布式的支持实时分析的数据存储系统（Data Store）。美国广告技术公司 MetaMarkets 于 2011 年创建了 Druid 项目，并且于 2012 年晚期开源了 Druid 项目。Druid 设计之初的想法就是为分析而生，它在处理数据的规模、数据处理的实时性方面，比传统的 OLAP 系统有了显著的性能改进，而且拥抱主流的开源生态，包括 Hadoop 等。多年以来，Druid 一直是非常活跃的开源项目。

Druid 的官方网站是 <http://druid.io>。

另外，阿里巴巴也曾创建过一个开源项目叫作 Druid（简称阿里 Druid），它是一个数据库连接池的项目。阿里 Druid 和本书讨论的 Druid 没有任何关系，它们解决完全不同的问题。

1.2 大数据分析和 Druid

大数据一直是近年的热点话题，随着数据量的急速增长，数据处理的规模也从 GB 级别增长到 TB 级别，很多图像应用领域已经开始处理 PB 级别的数据分析。大数据的核心目标

是提升业务的竞争力,找到一些可以采取行动的洞察 (Actionable Insight), 数据分析就是其中的核心技术, 包括数据收集、处理、建模和分析, 最后找到改进业务的方案。

最近一两年, 随着大数据分析需求的爆炸性增长, 很多公司都经历过将以关系型商用数据库为基础的数据平台, 转移到一些开源生态的大数据平台, 例如 Hadoop 或 Spark 平台, 以可控的软硬件成本处理更大的数据量。Hadoop 设计之初就是为了批量处理大数据, 但数据处理实时性经常是它的弱点。例如, 很多时候一个 MapReduce 脚本的执行, 很难估计需要多长时间才能完成, 无法满足很多数据分析师所期望的秒级返回查询结果的分析需求。

为了解决数据实时性的问题, 大部分公司都有一个经历, 将数据分析变成更加实时的可交互方案。其中, 涉及新软件的引入、数据流的改进等。数据分析的几种常见方法如图 1-1 所示。

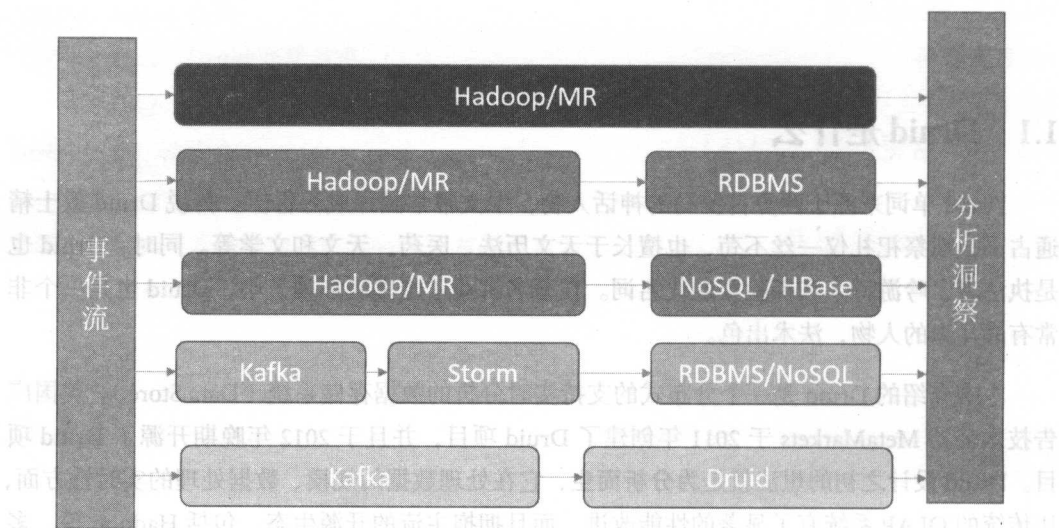


图 1-1 数据分析的几种常见方法

整个数据分析的基础架构通常分为以下几类。

- (1) 使用 Hadoop/Spark 的 MR 分析。
- (2) 将 Hadoop/Spark 的结果注入 RDBMS 中提供实时分析。
- (3) 将结果注入到容量更大的 NoSQL 中, 例如 HBase 等。
- (4) 将数据源进行流式处理, 对接流式计算框架, 如 Storm, 结果落在 RDBMS/NoSQL 中。
- (5) 将数据源进行流式处理, 对接分析数据库, 例如 Druid、Vertica 等。

1.3 Druid的产生

1.3.1 MetaMarkets 简介

Druid 诞生于 MetaMarkets 公司，简单介绍这家公司有助于更好地理解 Druid 创建的背景。这家公司致力于为在线媒体公司提供数据分析服务，客户包括在线广告公司、在线游戏开发商和社交媒体等。目前，互联网广告越来越多地采用程序化交易的方式，因此广告交易数据量大大增多，数据分析需求和场景也非常多。MetaMarkets 在这方面经验丰富，帮助广告交易平台（Ad Exchange）、供应方平台（Supply Side Platform）和需求方平台（Demand Side Platform）提供深度的数据分析服务。

广告程序化交易通常采用 iAB（Interactive Advertising Bureau）的 OpenRTB 标准，因此 MetaMarkets 的服务能力很容易扩展到多家广告公司。根据 MetaMarkets 数据，它们的数据分析平台的请求事件的峰值超过 3 百万/秒。另外，广告分析对于实时查询的性能要求非常高，Druid 就是 MetaMarkets 的核心数据处理平台，能够保证 99% 的数据查询在 1 秒内返回结果。在广告程序化交易模式下，广告流量将会发送到多个广告平台（Ad Network）或 DSP，DSP 需要根据数据和算法决定对广告流量进行出价。在整个过程中，数据处理和分析能力就是广告平台的核心竞争力。

在这个业务背景下，MetaMarkets 的工程师在早期的数据分析平台的设计上，经过两个阶段的努力后，最终决定自主开发 Druid 系统来满足业务需求。

第一阶段，基于 RDBMS 的查询分析。

MetaMarkets 最开始使用了 GreenPlum 社区版数据库，分析系统运行在亚马逊的云服务器上，机型为 M1.Large.EC2 Box。采用这种方式后，发现以下两个明显的问题。

- 很多全表扫描操作响应特别慢。例如，对于一个 3300 万的数据，计算总行数需要 3 秒时间。
- 所有列的处理方式相同。这意味着时间、维度、指标不做任何区分。因此在使用这些数据的时候，需要管理哪些是维度列，哪些是指标列。

对于第二个问题，可能只是增加了少量的系统管理成本，但是第一个问题对于分析平台确实是致命的。开发人员不得不通过一些预先计算（Pre-Caculate）或者缓存（Cache）机制，提高系统的使用性能。为了解决第一个问题，开发人员也尝试优化数据库的性能，或采用其他的数据库，例如 MySQL、InfoBright 和 PostgreSQL，但是情况并没有任何明显改善。

第二阶段，预计算结果放入 NoSQL 中。

经过第一阶段的尝试，开发人员得出一个结论：常规的关系型数据库管理系统（RDBMS）并不能满足实时大规模数据分析的性能需求。因此，他们在第二阶段的工作中，将 NoSQL 作为数据落地时的存储，利用 NoSQL 的高性能、可扩展特性解决数据分析的实时问题。

开发人员借鉴了 Twitter 的 Rainbird 类似的设计，开发了一套高性能的计数服务系统。Rainbird 是一款基于 Zookeeper、Cassandra、Scribe 和 Thrift 等技术的分布式实时统计系统，用于高性能的计数，支持获取最大值、最小值和平均值等功能，其中 Cassandra 用于存储数据内容。

MetaMarkets 也采用类似的理念设计了一套系统，该系统将所需要分析的指标，通过计数器的方法预先计算出来，并且放在 NoSQL 中，为数据分析提供实时能力。这种方法在维度比较少的时候，计算量不大，将数据放在 NoSQL 中，访问速度很快。

但是，随着维度的增加，预计算的计算量越来越大，当维度达到 11 个时，计算时间甚至超过了 24 小时。在后面的性能优化过程中，开发人员不得不限制预计算的维度，例如规定不能超过 5 个维度，只预先计算最重要的维度。但是，随着业务的发展，维度还在不断增加，计算量再次成为瓶颈。在最后的項目反思会中，大家都明确体会到这种方案的局限性，意识到当时做的决定是错误的。总结了这次失败的本质：Rainbird 这一类的高性能的计数器设计，并不适合通用数据分析的需求，特别是不能支持从不同角度或维度进行灵活的数据分析。

1.3.2 失败总结

在经历两次深刻的失败后，MetaMarkets 决定创建一个分布式的内存 OLAP 系统，用于解决以下两个核心问题。

- RDBMS 的查询太慢。
- 支持灵活的查询分析能力。

就这样，Druid 项目诞生了。

1.4 Druid 的三个设计原则

在设计之初，开发人员确定了三个设计原则（Design Principle）。

- （1）快速查询（Fast Query）：部分数据的聚合（Partial Aggregate）+ 内存化（In-Memory）+

索引 (Index)。

(2) 水平扩展能力 (Horizontal Scalability): 分布式数据 (Distributed Data) + 并行化查询 (Parallelizable Query)。

(3) 实时分析 (Realtime Analytics): 不可变的过去, 只追加的未来 (Immutable Past, Append-Only Future)。

1.4.1 快速查询 (Fast Query)

对于数据分析场景, 大部分情况下, 我们只关心一定粒度聚合的数据, 而非每一行原始数据的细节情况。因此, 数据聚合粒度可以是 1 分钟、5 分钟、1 小时或 1 天等。部分数据聚合 (Partial Aggregate) 给 Druid 争取了很大的性能优化空间。

数据内存化也是提高查询速度的杀手锏。内存和硬盘的访问速度相差近百倍, 但内存的大小是非常有限的, 因此在内存使用方面要精细设计, 比如 Druid 里面使用了 Bitmap 和各种压缩技术。

另外, 为了支持 Drill-Down 某些维度, Druid 维护了一些倒排索引。这种方式可以加快 AND 和 OR 等计算操作。

1.4.2 水平扩展能力 (Horizontal Scalability)

Druid 查询性能在很大程度上依赖于内存的优化使用。数据可以分布在多个节点的内存中, 因此当数据增长的时候, 可以通过简单增加机器的方式进行扩容。为了保持平衡, Druid 按照时间范围把聚合数据进行分区处理。对于高基数的维度, 只按照时间切分有时候是不够的 (Druid 的每个 Segment 不超过 2000 万行), 故 Druid 还支持对 Segment 进一步分区。

历史 Segment 数据可以保存在深度存储系统中, 存储系统可以是本地磁盘、HDFS 或远程的云服务。如果某些节点出现故障, 则可借助 Zookeeper 协调其他节点重新构造数据。

Druid 的查询模块能够感知和处理集群的状态变化, 查询总是在有效的集群架构中进行。集群上的查询可以进行灵活的水平扩展。Druid 内置提供了一些容易并行化的聚合操作, 例如 Count、Mean、Variance 和其他查询统计。对于一些无法并行化的操作, 例如 Median, Druid 暂时不提供支持。在支持直方图 (Histogram) 方面, Druid 也是通过一些近似计算的方法进行支持, 以保证 Druid 整体的查询性能, 这些近似计算方法还包括 HyperLoglog、DataSketches 的一些基数计算。

1.4.3 实时分析 (Realtime Analytics)

Druid 提供了包含基于时间维度数据的存储服务, 并且任何一行数据都是历史真实发生的事件, 因此在设计之初就约定事件一旦进入系统, 就不能再改变。

对于历史数据 Druid 以 Segment 数据文件的方式组织, 并且将它们存储到深度存储系统中, 例如文件系统或亚马逊的 S3 等。当需要查询这些数据的时候, Druid 再从深度存储系统中将它们装载到内存供查询使用。

1.5 Druid 的技术特点

Druid 具有如下技术特点。

- 数据吞吐量大。
- 支持流式数据摄入和实时。
- 查询灵活且快。
- 社区支持力度大。

1.5.1 数据吞吐量大

很多公司选择 Druid 作为分析平台, 都是看中 Druid 的数据吞吐能力。每天处理几十亿到几百亿的事件, 对于 Druid 来说是非常适合的场景, 目前已被大量互联网公司实践。因此, 很多公司选型 Druid 是为了解决数据爆炸的问题。

1.5.2 支持流式数据摄入

很多数据分析软件在吞吐量和流式能力上做了很多平衡, 比如 Hadoop 更加青睐批量处理, 而 Storm 则是一个流式计算平台, 真正在分析平台层面上直接对接各种流式数据源的系统并不多。

1.5.3 查询灵活且快

数据分析师的想法经常是天马行空, 希望从不同的角度去分析数据, 为了解决这个问题, OLAP 的 Star Schema 实际上就定义了一个很好的空间, 让数据分析师自由探索数据。数据量小的时候, 一切安好, 但是数据量变大后, 不能秒级返回结果的分析系统都是被诟病的

对象。因此，Druid 支持在任何维度组合上进行查询，访问速度极快，成为分析平台最重要的两个杀手锏。

1.5.4 社区支持力度大

Druid 开源后，受到不少互联网公司的青睐，包括雅虎、eBay、阿里巴巴等，其中雅虎的 Committer 有 5 个，谷歌有 1 个，阿里巴巴有 1 个。最近，MetaMarkets 之前几个 Druid 发明人也成立了一家叫作 Imply.io 的新公司，推动 Druid 生态的发展，致力于 Druid 的繁荣和应用。

接下来，我们一起来看看 Druid 的 Hello World 吧。

1.6 Druid 的 Hello World

1.6.1 Druid 的部署环境

Druid 系统用 Java 编写，目前支持 JDK 7 及以上版本。目前 Druid 论坛已经在开始讨论不再支持 Java 7，大部分公司都正在使用 Java 8 版本，旧版本的 Java 也在升级中，建议直接使用 Java 8 版本安装 Druid。

在操作系统方面，支持主流 Linux 和 Mac OS，不支持 Windows 操作系统。

内存配置越大越好，建议 8GB 以上。如果只用测试功能，4GB 内存也可运行。

1.6.2 Druid 的基本概念

1. 数据格式

Druid 在数据摄入之前，首先需要定义一个数据源（DataSource），这个 DataSource 有些类似数据库中表的概念。每个数据集合包括三个部分：时间列（TimeStamp）、维度列（Dimension）和指标列（Metric）。

（1）时间列

每个数据集合都必须有时间列，这个列是数据聚合的重要维度，Druid 会将时间很近的一些数据行聚合在一起。另外，所有的查询都需要指定查询时间范围。

(2) 维度列

维度来自于 OLAP 的概念，用来标识一些事件（Event），这些标识主要用于过滤或者切片数据，维度列的字段为字符串类型。例如，对于一次广告展现，我们可以将“哪个广告”、“哪个广告位置”、“计费的类型”、“广告主的类型”等作为广告展现的描述信息。随着业务分析的精细化，增加维度列也是一个常见的需求。

(3) 指标列

指标对应于 OLAP 概念中的 Fact，即用于聚合和计算的列。指标列字段通常为数值类型，计算操作通常包括 Count、Sum 和 Mean 等。指标通常是业务的关键量化指标，包括收入、使用时长等核心可度量和比较的指标。

数据格式样例如图 1-2 所示。

时间列	维度列			指标列	
时间	广告主	广告位置	是否点击	展现次数	展现价格
2011-01-01T01:01:35Z	A	位置1	Yes	20	20

图 1-2 数据格式样例

2. 数据摄入

Druid 提供两种数据摄入方式，如图 1-3 所示，其中一种是实时数据摄入；另一种是批处理数据摄入。

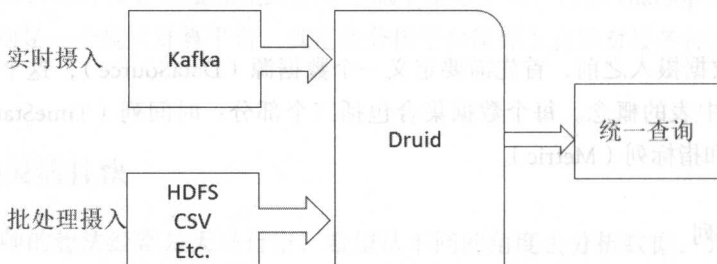


图 1-3 两种数据摄入方式

3. 数据查询

在数据查询方面, Druid 原生查询是采用 JSON 格式, 通过 HTTP 传送。Druid 不支持标准的 SQL 语言查询, 因为有些 SQL 语言查询并不适用于 Druid 现在的设计。为了简化查询的解析, 采用自己定义的 JSON 格式, 方便内外部处理。随着 Druid 应用越来越广泛, 支持标准 SQL 的需求变得越来越重要, 一些开源生态项目正在向这个方面努力, 例如 Imply.io 的 PlyQL 等。

对于 Druid 的查询访问, 除了原生 Java 客户端支持外, 也出现了很多支持不同语言客户端访问的开源项目, 例如 Python、R、JavaScript 和 Ruby 等。

下面是一个用 JSON 表达的查询例子, 该查询中指定了时间范围、聚合粒度、数据源等。

```
{
  "queryType": "timeseries",
  "dataSource": "sample_datasource",
  "granularity": "day",
  "aggregations": [
    { "type": "longSum", "name": "result_name", "fieldName": "field_name" }
  ],
  "intervals": ["2012-01-01T00:00:00.000/2012-01-04T00:00:00.000"],
  "context": { "skipEmptyBuckets": "true" }
}
```

1.7 系统的扩展性

Druid 是一个分布式系统, 采用 Lambda 架构, 将实时数据和批处理数据合理地解耦。实时数据处理部分是面向写多读少的优化, 批处理部分是面向读多写少的优化。整个分布式系统采用 Shared nothing 的结构, 各个节点都有自己的计算和存储能力。整个系统使用 Zookeeper 进行协调, 另外还使用了 MySQL 提供一些元数据的存储。

在雅虎, 已经验证过数百台 Druid 集群的规模。国内的阿里巴巴、小米、优酷等公司也有大规模部署 Druid 的成功经验。

MetaMarkets 曾经介绍过它的一些扩展能力的数字。

- 每天 1000 亿的事件
- 每秒超过 300 万的事件
- 超过 100PB 的原始数据

- 超过 50000 亿的总数据
- 上千用户的每秒查询峰值
- 数万个处理器核

1.8 性能指标

性能测试涉及太多的因素，包括测试数据源、访问模式、机器配置和部署等，因此很难有统一的业界标准。在参考资料中，提到了根据 TPC-H 标准数据集的性能评测结果。由于 Druid 不能支持所有的查询，所以测试只覆盖了支持的查询场景。

对于 1GB 的数据和 100GB 的数据，从查询性能来看，Druid 基本完胜 MySQL，查询时间缩小了 5~10 倍。随着数据量的增大，这种性能改善的程度会越来越大。

如图 1-4 所示是 100GB 规模的 Druid 测试数据，从测试结果来看，性能提升也是非常明显的。

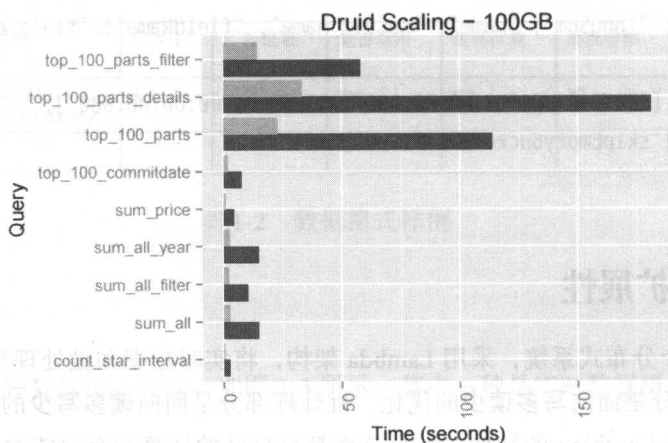


图 1-4 100GB 规模的 Druid 测试数据

1.9 Druid 的应用场景

从技术定位上看，Druid 是一个分布式的数据分析平台，在功能上也非常像传统的 OLAP 系统，但是在实现方式上做了很多聚焦和取舍，为了支持更大的数据量、更灵活的分布式部署、更实时的数据摄入，Druid 舍去了 OLAP 查询中比较复杂的操作，例如 JOIN 等。相比传统数据库，Druid 是一种时序数据库，按照一定的时间粒度对数据进行聚合，以加快分析查询。

在应用场景上, Druid 从广告数据分析平台起家, 已经广泛应用在各个行业和很多互联网公司中, 下面将介绍一些使用 Druid 的公司, 最新列表可以访问 <http://druid.io/druid-powered.html>。

1.9.1 国内公司

1. 腾讯

腾讯是一家著名的社交互联网公司, 其明星产品如 QQ、微信有着上亿级别庞大的用户量。在 2B 业务领域, 作为中国领先的 SaaS 级社会化客户关系管理平台, 腾讯企点采用了 Druid 用于分析大量的用户行为, 帮助提升客户价值。

2. 阿里巴巴

阿里巴巴是世界领先的电子商务公司。阿里搜索组使用 Druid 的实时分析功能来获取用户的交互行为。

3. 新浪微博

新浪微博是中国领先的社交平台。新浪微博的广告团队使用 Druid 构建数据洞察系统的实时分析部分, 每天处理数十亿的消息。

4. 小米

小米是中国领先的专注智能产品和服务的移动互联网公司。Druid 用于小米统计的部分后台数据收集和分析; 另外, 在广告平台的数据分析方面, Druid 也提供了实时的内部分析功能, 支持细粒度的多维度查询。

5. 滴滴打车

滴滴打车是世界领先的交通平台。Druid 是滴滴实时大数据处理的核心模块, 用于滴滴的实时监控系統, 支持数百个关键业务指标。通过 Druid, 滴滴能够快速得到各种实时的数据洞察。

6. 优酷土豆

优酷土豆是中国领先的互联网视频公司, Druid 用于其广告平台的数据处理和分析。

7. 蓝海讯通 (OneAPM)

蓝海讯通是中国领先的应用性能管理 (APM) 的技术服务公司。Druid 用于实时的应用监测数据收集, 并且提供给用户灵活的查询功能。

8. YeahMobi

YeahMobi 是一家专注移动互联网的营销公司。Druid 用于广告数据的实时分析, 包括转化率、IP 分布情况和收入等维度的分析。

1.9.2 国外公司

1. 雅虎 (Yahoo!)

雅虎是全球领先的互联网公司, 它也是最早一批深度使用 Druid 的公司。雅虎曾经维护着世界上最大的 Hadoop 集群, 但是 Hadoop 集群无法处理实时交互查询, 无法支持实时数据摄入, 无法灵活支持每日几百亿的事件。在尝试很多工具之后, 最后他们还是深度拥抱了 Druid。

Druid 用于数据收集, 为管理层提供仪表盘, 为客户提供实时的数据查询功能。另外, 雅虎也深度参与这个项目的开发当中, 不少 Committer 都来自雅虎公司, 雅虎也在和社区紧密合作推动 Druid 的发展。

2. PayPal

PayPal 是世界领先的互联网支付公司。2014 年年初, PayPal 的 Tracking Platform 组采用了 Druid 处理每天 70~100 亿条的记录数据, 查询的响应时间非常理想。如今 Druid 在 PayPal 已经有一个非常大的集群, 为业务分析组提供了各种各样的数据分析支持。

3. eBay

eBay 是互联网电子商务的领先公司。eBay 使用 Druid 聚合多个数据源, 用于用户行为分析, 数据量超过 10 万消息/秒。同时在查询方面, Druid 提供了一个自由组合的条件查询功能, 支持商业分析的场景。另外, eBay 的云平台组也在 2016 年 2 月开源了一个 embedded-druid 项目, 针对单机 JVM 进程提供嵌入式的 Druid 解决方案, 它不涉及复杂节点的部署, 单节点, 高性能。

4. Hulu

Hulu 是美国领先的互联网视频公司。Hulu 使用 Druid 构建数据分析平台，进行实时的用户行为和应用数据分析。

5. 思科 (Cisco)

思科是世界领先的通信技术公司之一，使用 Druid 对网络数据流进行实时的数据分析。

此外，还有很多使用 Druid 的公司，广告公司还包括 Criteo、LDMobile、MetaMarkets、PubNative、VideoAmp 和 GumGum 等，数据分析公司还包括 Optimizely、Monetate、AppsFlyer 和 Archive-it.org 等，其他公司还包括 AirBnb、DripStat 等。

总结这些公司的使用场景，Druid 确实提供了一个相对通用的数据分析平台，起源于广告数据分析，但广泛应用于用户行为分析、网络数据分析等领域。不仅仅有互联网公司在使用，也有很多老牌公司在使用，例如 Cisco、SK Telecom 等。大部分公司都看中了 Druid 的大数据量处理能力、数据实时性和秒级数据查询功能。

1.10 小结

Druid 从出生到繁荣，很单纯和专注地解决了大规模数据的分析问题，实时性是它的特色之一。

在 Hadoop 和 Spark 大生态繁荣圈中，Druid 能获得众多公司的青睐，异军突起，总结起来有三个原因。

第一，Druid 不断拥抱各种主流开源生态，例如 Hadoop、Spark 等。

第二，围绕着 Druid，已经开始出现一些项目，完善了 Druid 的生态系统，其中包括数据的摄入、客户端的访问、数据查询的便利和数据的可视化等。

第三，出现了专业的 Druid 技术服务公司，例如 Implied.io 等，帮助企业客户更好地认识 Druid，使用和优化 Druid，以解决客户的数据分析需求。

我们非常高兴看到最近几年 Druid 获得的长足快速发展，并且解决了很多公司在业务上的数据分析痛点。在技术上 Druid 采用分布式、Shared nothing 结构，提供了实时和离线批量数据摄入的能力，通过数据的预聚合、优化存储结构和内存使用，保证了在查询分析方面的高效率。在应用场景上 Druid 满足了 OLAP 的核心功能，在社区推广方面 Druid 也同样值得称赞。

参考资料

Druid 主页: <http://druid.io>

Druid 应用公司列表: <http://druid.io/druid-powered.html>

Druid 的技术论文: <http://static.druid.io/docs/druid.pdf>

第2章

数据分析及相关软件

本章我们将介绍数据分析及相关软件，以帮助读者了解数据分析软件的一些分类，找到适合自己场景的分析软件，并且和 Druid 做一些技术上的比较。

2.1 数据分析及相关概念

数据分析（Data Analytics）从来都不是一个寂寞的领域，每一个时代都赋予数据分析更丰富的内容和精尖的技术。

数据分析是指通过数据的收集，进行数据处理和分析，将数据整理成有用的信息，包括有价值的洞察和可以付之于行动的建议。数据分析的目的就是帮助我们把数据（Data）变成信息（Information），再从信息变成知识（Knowledge），最后从知识变成智慧（Wisdom）。其过程如图 2-1 所示。

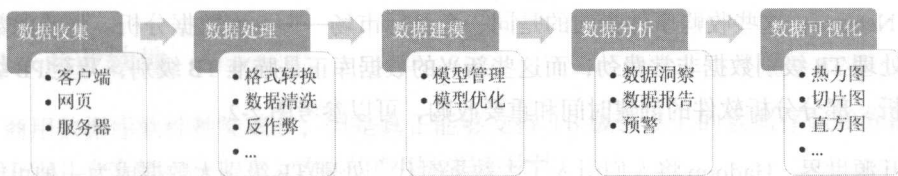


图 2-1 数据分析的整个过程

在数据分析领域，商务智能（Business Intelligence, BI）、数据挖掘（Data Mining, DM）、联机分析处理（On-Line Analytical Processing, OLAP）等概念在名称上和数据分析方面非常

接近，容易混淆，下面做个简单介绍。

商务智能：商务智能（BI）是在商业数据上进行价值挖掘的过程。商务智能的历史很长，很多时候会特别指通过数据仓库技术进行业务报表制作和分析的过程，在分析方法上通常使用聚合（Aggregation）、分片（Slice）等方式进行数据处理。在技术上，商务智能包括 ETL（数据的抽取、转换、加载）、数据仓库（Data Warehouse）、OLAP（联机分析处理）、数据挖掘（Data Mining）等技术。

数据挖掘：数据挖掘（DM）是指在大量数据中自动搜索隐藏于其中有着特殊关系（属于 Association rule learning）信息的过程。很多年前，它一直是一个热门的研究生专业，直到信息检索专业的出现。

联机分析处理：联机分析处理（OLAP）是一种建立数据系统的方法，其核心思想即建立多维度的数据立方体，以维度（Dimension）和度量（Measure）为基本概念，辅以元数据实现可以钻取（Drill-down/up）、切片（Slice）、切块（Dice）等灵活、系统和直观的数据展现。

数据分析既是一门艺术（所谓艺术就是结合技术、想象、经验和意愿等综合因素的平衡和融合），也是一个经验和想象力的融合（它涉及数学算法、统计分析、工具和软件工程的一种结合，以解决业务的问题为目标，帮助人们从数据中获得智慧）。

2.2 数据分析软件的发展

数据分析软件市场一直以来都很活跃，从两个视角来看，一个是商业软件市场，充满了大鱼吃小鱼的故事；另一个是开源数据存储处理软件，在互联网精神和开源情怀的引导下，各种专业领域的开源软件日益壮大，通用数据存储系统也不断升级。

2011 年，Teradata 收购了 Aster Data 公司，同年惠普收购了实时分析平台 Vertica 等；2012 年，EMC 收购了数据仓库软件厂商 GreenPlum；2013 年，IBM 宣布以 17 亿美元收购数据分析公司 Netezza；这些收购事件指向的是同一个目标市场——大数据分析。传统的数据库软件，处理 TB 级别数据非常费劲，而这些新兴的数据库正是瞄准 TB 级别，乃至 PB 级别的数据分析。部分分析软件的创建时间和重要收购，可以参考图 2-2。

在开源世界，Hadoop 将人们引入了大数据时代，处理 TB 级别大数据成为一种可能，但实时性能一直是 Hadoop 的一个伤痛。2014 年，Spark 横空出世，通过最大利用内存处理数据的方式，大大改进了数据处理的响应时间，快速发展出一个较为完备的生态系统。另外，大量日志数据都存放在 HDFS 中，如何提高数据处理性能，支持实时查询功能则成为不少开源数据软件的核心目标。例如，Hive 利用 MapReduce 作为计算引擎，Presto 自己开发计算引

引擎，以及 Druid 自己开发索引和计算引擎等，都是为了一个目标：处理更多的数据，获取更高的性能。

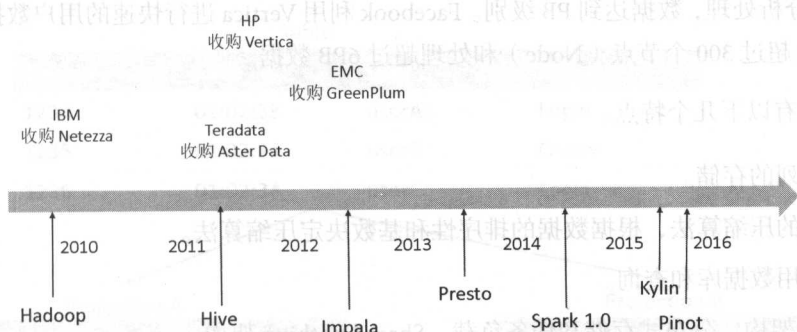


图 2-2 部分分析软件的创建时间和重要收购

2.3 数据分析软件的分类型

为了全面了解数据分析软件的分类型，本节将介绍一些适合大数据分析的存储数据库，或者面向大数据分析，适用于 TB 级别以上的数据存储和分析任务。接下来，按照以下几个分类型来介绍。在介绍完每种软件的技术特点后，我们会与 Druid 做一些简单的技术性比较。

- 商业数据库
- 开源时序数据库
- 开源计算框架
- 开源数据分析软件
- 开源 SQL on Hadoop
- 云端数据分析 SaaS

2.3.1 商业软件

商用数据库软件种类繁多，但是真正能够支持 TB 级别以上的数据存储和分析并不多，这里介绍几个有特点的支持大数据的商用数据库软件。

1. HP Vertica

Vertica 公司成立于 2005 年，创立者为数据库巨擘 Michael Stonebraker（2014 年图灵奖获得者，INGRES、PostgreSQL、VoltDB 等数据库发明人）。2011 年 Vertica 被惠普收购。Vertica

软件是能提供高效数据存储和快速查询的列存储数据库实时分析平台，还支持大规模并行处理（MPP）。产品广泛应用于高端数字营销、互联网客户（比如 Facebook、AOL、Twitter、Groupon）分析处理，数据达到 PB 级别。Facebook 利用 Vertica 进行快速的用户数据分析，据称 Facebook 超过 300 个节点（Node）和处理超过 6PB 数据。

Vertica 有以下几个特点。

- 面向列的存储。
- 灵活的压缩算法，根据数据的排序性和基数决定压缩算法。
- 高可用数据库和查询。
- MPP 架构，分布式存储和任务负载，Shared nothing 架构。
- 支持标准的 SQL 查询、ODBC/JDBC 等。
- 支持 Projection（数据投射）功能。

如图 2-3 所示。

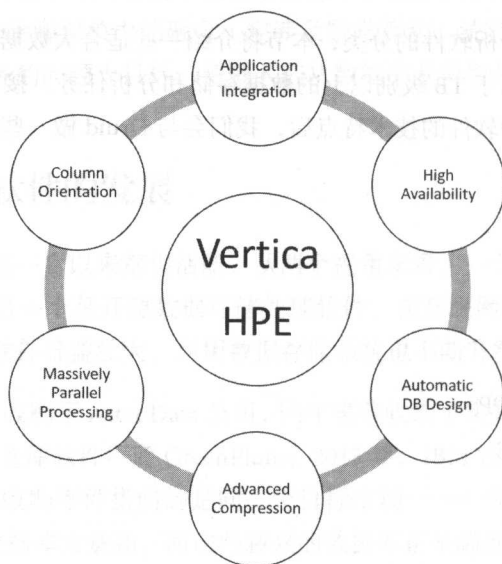


图 2-3 Vertical 的特点

相比 Druid，Vertica 支持更多的企业级特性，例如 SQL、JDBC 等标准，支持数据拆分的 Projection 功能，但缺少 Druid 的数据部分聚合，缺少数据的实时摄入功能。另外一个区别是 Vertica 不使用 Index，而是尝试利用行程编码（Run-Length Encoding），以及其他的压缩技术和产生优化的存储结构。

接下来介绍一下 Projection 技术, 它的原理是将数据的某些列提取出来进行专门的存储, 以加快后期的数据访问速度, 同一个列可以在不同的 Projection 中, 如图 2-4 所示。

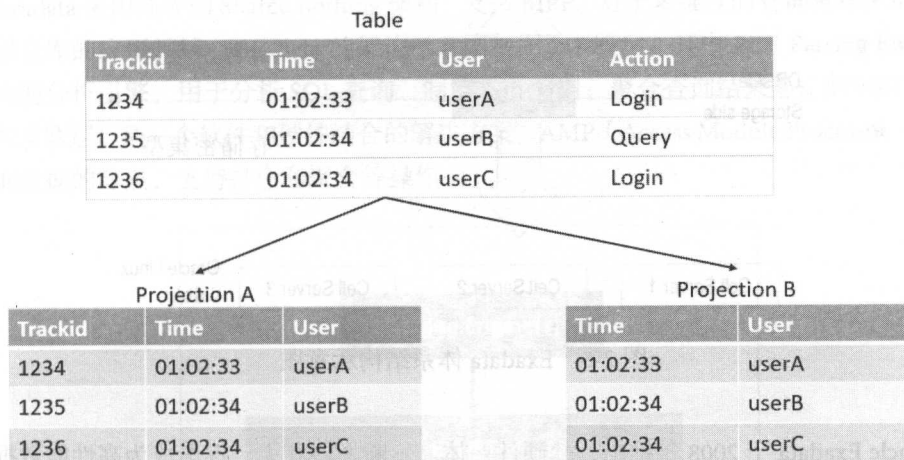


图 2-4 Projection 示意图

2. Oracle Exadata

Oracle Exadata 是数据库发展史上的一个传奇, 它是数据库软件和最新硬件的完美结合。它提供最快、最可靠的数据库平台, 不仅支持常规的数据库应用, 也支持联机分析处理 (OLAP) 和数据仓库 (DW) 的场景。

Exadata 采用了多种最新的硬件技术, 例如 40Gb/s 的 InfiniBand 网络 (InfiniBand 是超高速的专用数据网络协议, 主要专用硬件支持)、3.2TB PCI 闪存卡 (每个闪存卡都配有 Exadata 智能缓存, X6 型号)。Oracle Exadata 数据库云服务器允许将新一代服务器和存储无缝部署到现有的 Oracle Exadata 数据库云服务器中。它包含两套子系统, 一套处理计算密集型的查询, 一套处理存储密集型的查询, Exadata 能够做到智能查询分配。

Exadata 体系结构示意图如图 2-5 所示。

Oracle ExaData 有如下几个技术特点。

- 采用 InfiniBand 高速网络, 采用极速闪存方案。
- 全面的 Oracle 数据库兼容性。
- 针对所有数据库负载进行了优化, 包括智能扫描。
- Oracle Exadata 支持混合列压缩。

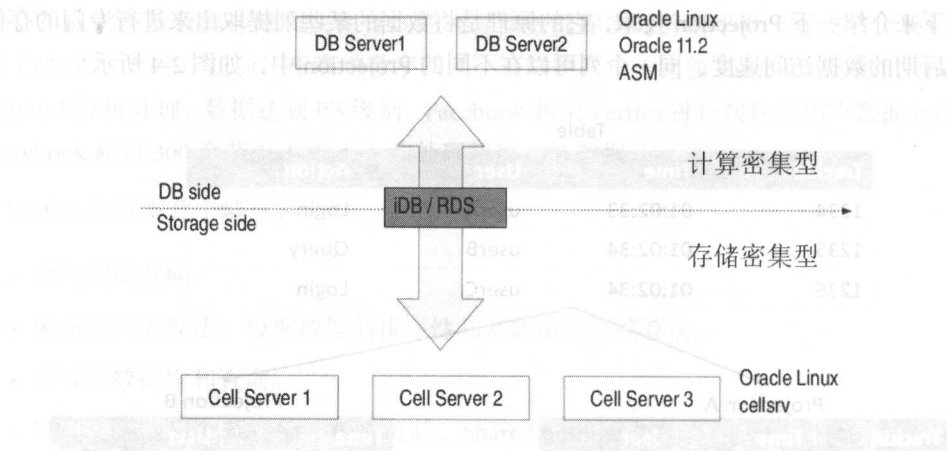


图 2-5 Exadata 体系结构示意图

Oracle Exadata 于 2008 年推出，软硬件一体，不断发展壮大，逐渐成为高性能数据库的代名词。其发展历程如图 2-6 所示。

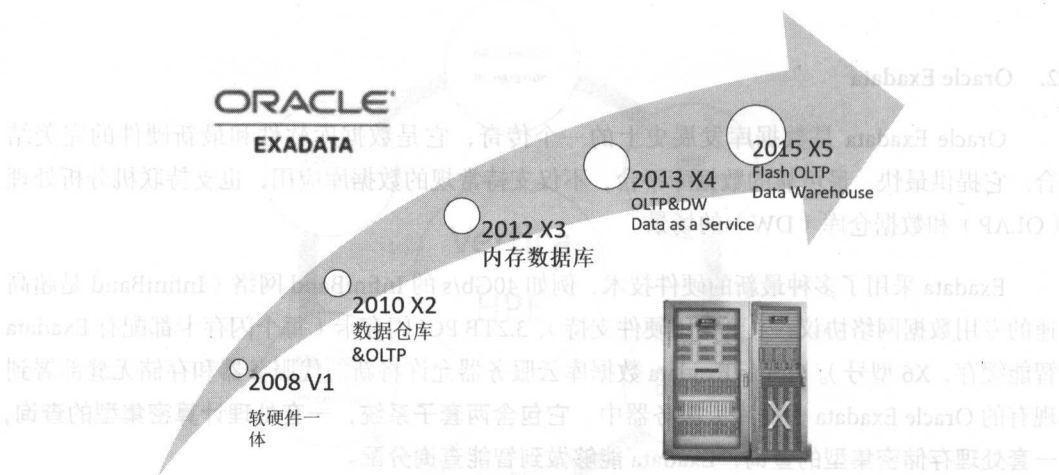


图 2-6 Oracle Exadata 的发展历程

相比 Druid，Oracle Exadata 提供了大而全的企业解决方案，通过硬件加速查询，在处理 TB 级别的数据量上有很大的性能优势。对于处理 PB 级别的数据量，Exadata 可能会有些力不从心，当然成本也是一个大问题。

Oracle Exadata 混合列存储是介于行存储和列存储之间的一个方案，主要思想是对列进行分段处理，每一段都使用列式存储放在 Block 中，而后按照不同的压缩策略进行处理。

3. Teradata

Teradata（天睿）公司是专注于大数据分析、数据仓库和整合营销管理解决方案的供应商。Teradata 采用纯粹的 Shared nothing 架构，支持 MPP。对于多维度的查询更加灵活，专注于数据仓库的应用领域。Teradata 的架构示意图如图 2-7 所示，其中 PE（Parsing Engine(s)）是查询的分析引擎，用于分析 SQL 查询、制定查询计划、聚合查询结果等；BYNET 是一个高速的互联层，是一个软件和硬件结合的解决方案。AMP（Access Module Processor）是存储和查询数据的节点，支持排序和聚合等操作。

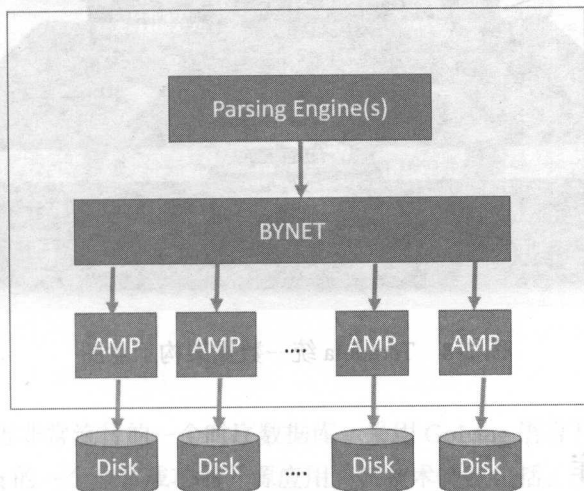


图 2-7 Teradata 的架构示意图

另外，Teradata 还提出了一个统一的数据分析框架，如图 2-8 所示，其中包括两个核心产品，一个是 Teradata 数据仓库，另一个是 Teradata Aster 数据分析产品。这两个产品分别走不同的路线：Teradata 是传统数据仓库，满足通用的数据需求；Aster 实际上是一种基于 MapReduce 的数据分析解决方案，可以支持更加灵活的数据结构的处理，例如非结构化数据的处理。

Teradata 提供了一个完整的数据解决方案，包括数据仓库和 MapReduce。Druid 则聚焦在数据仓库中的时序数据库问题上。

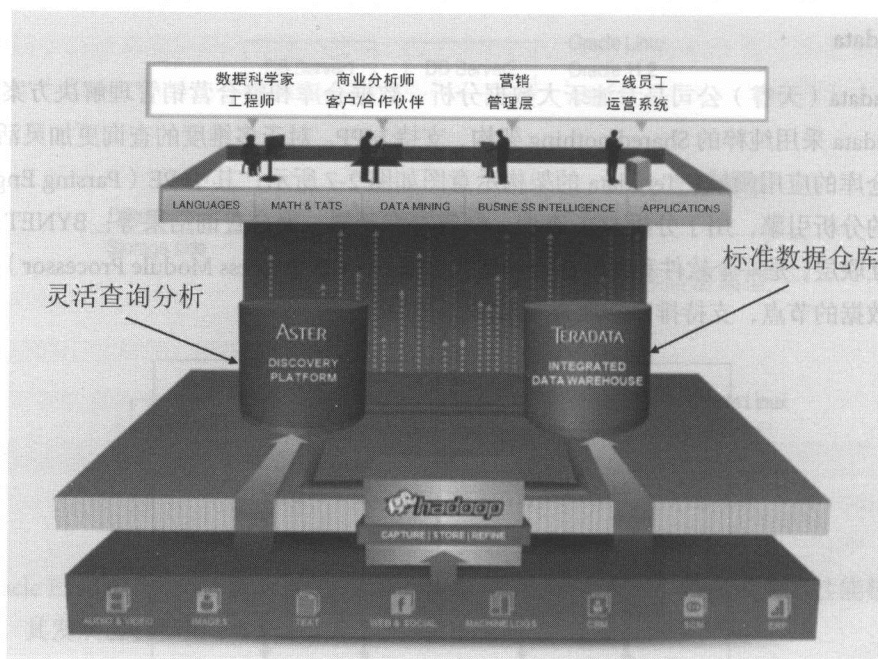


图 2-8 Teradata 统一数据架构示意图

2.3.2 时序数据库

时序数据库用于记录过去时间的各个数据点的信息，典型的应用是服务器的各种性能指标，例如 CPU、内存使用情况等。目前时序数据库也广泛应用于各种传感器的数据收集分析工作中，这些数据的收集都有一个特点，就是对时间的依赖非常大，每天产生的数据量非常大，因此写入的量非常大，一般的关系型数据库无法满足这些场景。因此，时序数据库在设计上需要支持高吞吐、高效数据压缩，支持历史查询、分布式部署等。虽然 Druid 更加接近数据仓库的角色，但是在很多特性上它也属于一种时序数据库。

1. OpenTSDB

OpenTSDB 是一个开源的时序数据库，支持存储数十亿的数据点，并提供精确查询功能。它采用 Java 语言编写，通过基于 HBase 的存储实现横向扩展。OpenTSDB 广泛用于服务器性能的监控和度量，包括网络和服务器、传感器、IoT、金融数据的实时监控领域。OpenTSDB 应用于很多互联网公司的运维系统中，例如 Pinterest 公司有超过 100 个节点的部署，雅虎公司也有超过 50 个节点的部署。它的设计思路是利用 HBase 的 Key 存储一些 Tag 信息，将同一个小时数据放在一行存储，提高了查询速度。其设计架构示意图如图 2-9 所示。

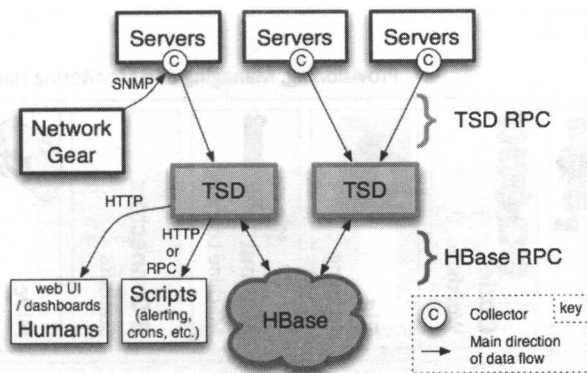


图 2-9 OpenTSDB 设计架构示意图

相比 Druid, OpenTSDB 在运维监控领域有长时间的应用,两者在技术上的区别比较明显,OpenTSDB 通过预先定义好维度 Tag 等,采用精巧的数据组织方式放入 HBase 中,利用 HBase 的 KeyRange 进行快速查询,在任意维度的组合查询下,OpenTSDB 查询效率会明显降低。

2. InfluxDB

InfluxDB 是最近非常流行的一个时序数据库,采用 GoLang 语言开发,目前它的社区非常活跃,是 GoLang 的一个非常成功的开源应用。其技术特点包括:支持任意数量的列,支持方便、强大的查询语言,集成了数据采集、存储和可视化功能。它支持高效存储,使用高压缩比的算法等。在早期设计中,使用 LevelDB 作为存储,后来改成 Time Series Merge Tree 作为内部存储,支持与 SQL 类似的查询语言。

相比 Druid, InfluxDB 的设计更加针对 OpenTSDB 的一些场景,内部存储结构和 Druid 也有非常大的差别。

2.3.3 开源分布式计算平台

1. Hadoop

Hadoop 是一个分布式系统基础架构,由 Apache 基金会开发。用户可以在不了解分布式底层细节的情况下,开发分布式程序,充分利用集群的威力高速运算和存储。Hadoop 实现了一个分布式文件系统 (Hadoop Distributed File System, HDFS)。除了文件存储, Hadoop 还有最完整的大数据生态,包括机器管理、NoSQL KeyValue 存储 (如 HBase)、协调服务 (Zookeeper 等)、SQL on Hadoop (Hive) 等。其整体生态如图 2-10 所示。

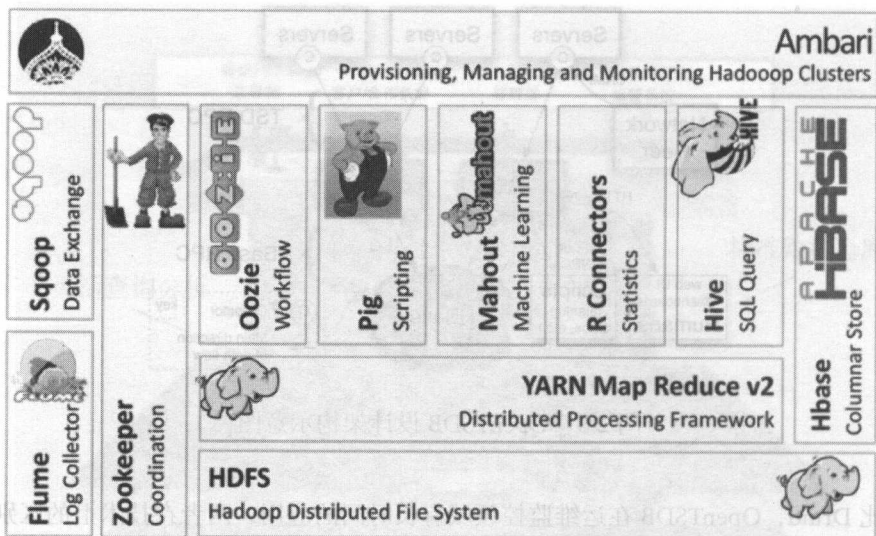


图 2-10 Hadoop 软件生态图

Hadoop 基于可靠的分布式存储, 通过 MapReduce 进行迭代计算, 查询批量数据。Hadoop 是高吞吐的批处理系统, 适合大型任务的运行, 但在对任务响应时间和实时性有严格要求的需求方面 Hadoop 并不擅长。

Druid 正好是 Hadoop 的一个有利补充, 它提供了一套非常实时的方案, 并可利用 HDFS 作为其深度存储 (Deep Storage) 数据文件的一种解决方案。另外, Druid 也全面拥抱 Hadoop 生态并能够对接很多 Hadoop 生态中的数据源。

2. Spark

Spark 是 UC Berkeley AMP lab 开源的类 Hadoop MapReduce 通用的并行计算框架, 同样也是基于分布式计算, 拥有 Hadoop MapReduce 的所有优点; 不同的是, Spark 任务的中间计算结果可以缓存在内存中, 这样迭代计算时可以从内存直接获取中间结果而不需要频繁读写 HDFS, 因此 Spark 的运行速度更快, 适用于对性能有要求的数据挖掘与数据分析场景。Spark 生态组件也较多, 核心组件如图 2-11 所示。

Spark 是实现弹性分布式数据集概念的计算集群系统, 可以看作是商业分析平台。RDD 能复用持久化到内存中的数据, 从而为迭代算法提供更快计算速度。这对一些工作流例如机器学习格外有用, 比如有些操作需要重复执行很多次才能达到结果的最终收敛。同时, Spark 也提供了大量的算法用来查询和分析数据, 其开发语言采用 Scala, 因此直接在上面做数据处理和分析, 开发成本会比较高, 适合非结构化的数据查询处理。

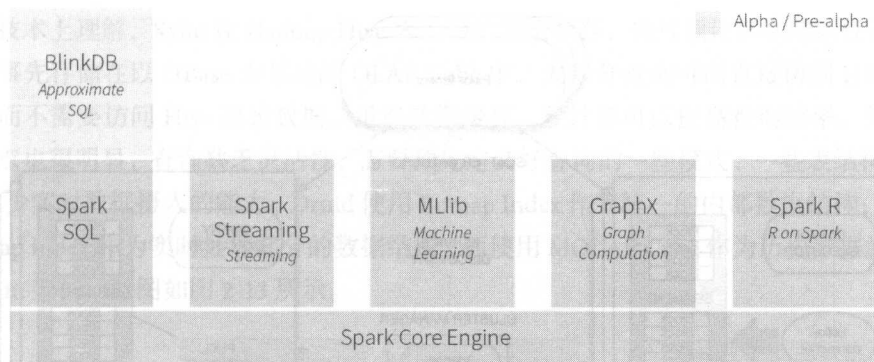


图 2-11 Spark 的核心组件

Druid 的核心是通过数据预先聚合提高查询性能，针对预先定义好的 Schema，因此适合实时分析的场景，结果返回时间在亚秒级。Spark 可以对任何 Schema 进行灵活操作，适合处理规模更大的批处理任务。

2.3.4 开源分析数据库

1. Pinot

如果要找一个与 Druid 最接近的系统，那么非 LinkedIn Pinot 莫属。Pinot 是 LinkedIn 公司于 2015 年年底开源的一个分布式列式数据存储系统。LinkedIn 在开源界颇具盛名，大名鼎鼎的 Kafka 就来源于 LinkedIn，因此 Pinot 在推出后就备受关注和追捧。

Pinot 的技术特点如下。

- 一个面向列式存储的数据库，支持多种压缩技术
- 可插入的索引技术——Sorted Index、Bitmap Index 和 Inverted Index
- 可以根据 Query 和 Segment 元数据进行查询和执行计划的优化
- 从 Kafka 的准实时数据灌入和从 Hadoop 的批量数据灌入
- 类似于 SQL 的查询语言和各种常用聚合
- 支持多值字段
- 水平扩展和容错

在架构上，Pinot 也采用了 Lambda 架构，如图 2-12 所示，将实时数据流和批处理数据分开处理。其中 Realtime Node 处理实时数据查询，Historical Node 处理历史数据。

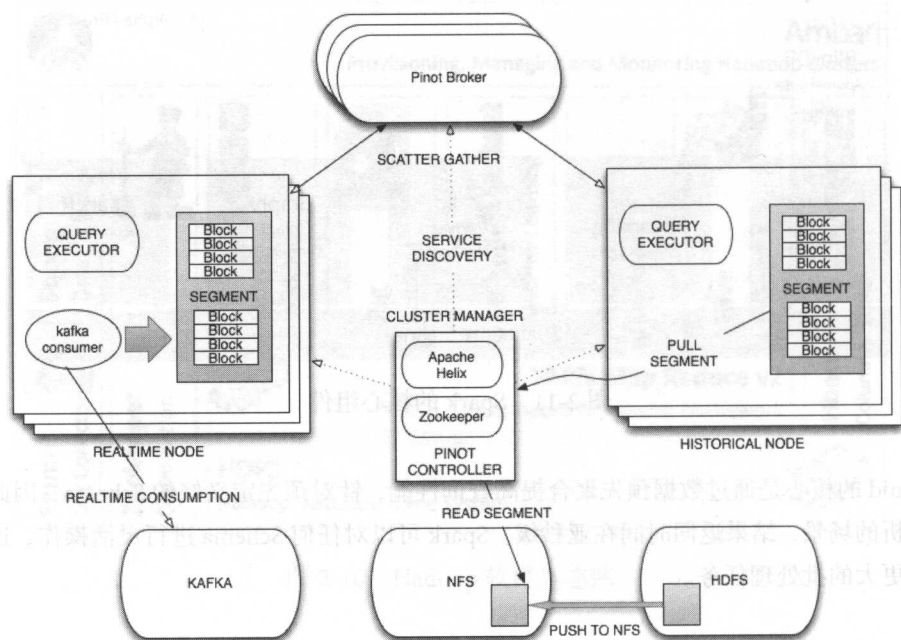


图 2-12 Pinot 架构示意图

相比 Druid，Pinot 还比较年轻，相关生态支持力度小，但像很多刚开放的开源软件一样，目前 Pinot 社区也非常活跃。Pinot 的索引结构支持更多的索引格式，如 Forwarded Index、Sorted Forwarded Index 和 Multi Value Forward Index。当列的基数较少（例如少于 10K）时，此列就会采用字典编码（Dictionary Encoding），而后对列内的数据进行压缩存储。Druid 支持比较单一的 Bitmap 方式的索引。对于查询语言来说，Pinot 的查询语法更接近 SQL 方式。Pinot 的集群管理也使用 Apache 开源软件 Helix，性能更可靠，可管理性更强。总体来说，Pinot 的整体设计比 Druid 更加完整和系统化，但是该技术刚刚开源，离成熟应用还有很长一段路要走。

2. Kylin

Kylin 是 Apache 开源的一个分布式分析引擎，提供了在 Hadoop 之上的 SQL 查询接口及多维分析（OLAP）能力，可以支持超大规模数据。它最初由 eBay 公司开发并于 2015 年贡献至开源社区。Kylin 能在亚秒内查询巨大的 Hive 表。

Kylin 的优势很明显，它支持标准的 ANSI SQL 接口，可以复用很多传统的数据集成系统，支持标准的 OLAP Cube，查询数据更加方便，与大量 BI 工具无缝整合。另外，它提供了很多管理功能，例如 Web 管理、访问控制、支持 LDAP、支持 HyperLoglog 的近似算法。

从技术上理解, Kylin 在 Hadoop Hive 表上做了一层缓存, 通过预计算和定期任务, 把很多数据事先存储在以 HBase 为基础的 OLAP Cube 中, 大部分查询可以直接访问 HBase 获取结果, 而不需要访问 Hive 原始数据。虽然数据缓存、预计算可以提高查询效率, 但这种方式的缺点也很明显, 查询缺乏灵活性, 需要预先定义好查询的一些模式、一些表结构。目前, Kylin 缺少实时数据摄入的能力。Druid 使用 Bitmap Index 作为统一的内部数据结构; Kylin 使用 Bitmap Index 作为实时处理部分的数据结构, 而使用 MOLAP Cube 作为历史数据的数据结构。Kylin 架构示意图如图 2-13 所示。

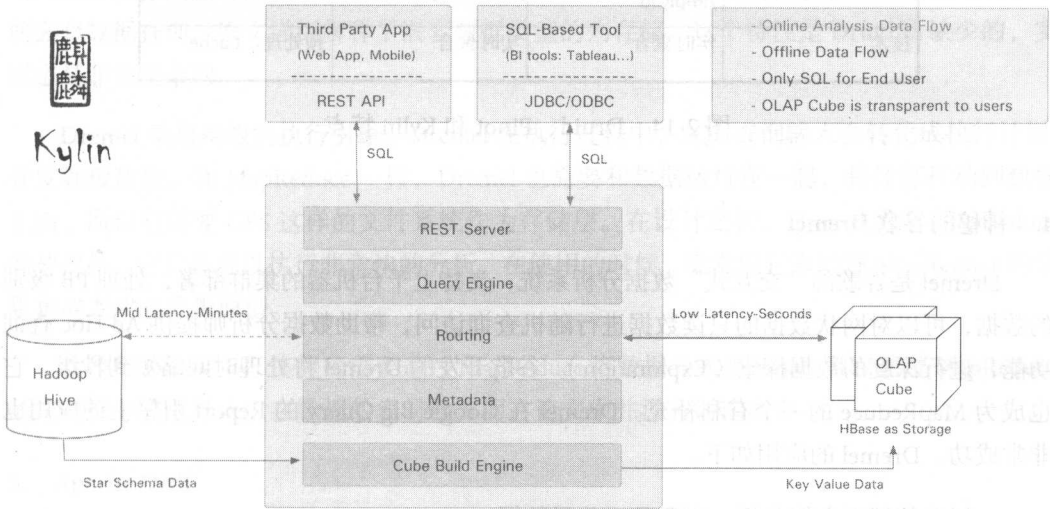


图 2-13 Kylin 架构示意图

另外, 在 Kylin 开发成员中有很多开发人员来自中国, 因此在 PMC 成员中中国人占了大部分, 所以使用 Kylin 很容易得到很好的中文支持。Kylin 的愿景就是创建一个分布式的高可扩展的 OLAP 引擎。

3. Druid、Pinot 和 Kylin 比较

Druid、Pinot 和 Kylin 是数据分析软件选型经常碰到的问题。Druid 和 Pinot 解决的业务问题非常类似。Pinot 架构的设计比较规范, 系统也比较复杂, 由于开源时间短, 社区的支持力度弱于 Druid。Druid 的设计轻巧, 代码库也比较容易懂, 支持比较灵活的功能增强。Kylin 的最大优势是支持 SQL 访问, 可以兼容传统的 BI 工具和报表系统, 在性能上没有太大优势。

这几个软件的特点如图 2-14 所示。

	Druid	Pinot	Kylin
使用场景	实时处理分析	实时处理分析	OLAP分析引擎
开发语言	Java	Java	Java
接口协议	JSON	JSON	OLAP/JDBC
发布时间	2011年	2015年	2015年
资助者	MetaMarkets /Imply.io	LinkedIn	eBay
技术	实时聚合	实时聚合	预处理, Cache

图 2-14 Druid、Pinot 和 Kylin 特点

4. 神秘的谷歌 Dremel

Dremel 是谷歌的“交互式”数据分析系统，支持上千台机器的集群部署，处理 PB 级别的数据，可以对网状数据的只读数据进行随机查询访问，帮助数据分析师提供 Ad Hoc 查询功能，进行深度的数据探索（Exploration）。谷歌开发的 Dremel 将处理时间缩短到秒级，它也成为 MapReduce 的一个有利补充。Dremel 在 Google Big Query 的 Report 引擎上的应用也非常成功。Dremel 的应用如下。

- 抓取的网页文档分析，主要是一些元数据
- 追踪 Android 市场的所有安装数据
- 谷歌产品的 Crash 报告
- 作弊（Spam）分析
- 谷歌分布式构建（Build）系统中的测试结果
- 上千万的磁盘 I/O 分析
- 谷歌数据中心中任务的资源分析
- 谷歌代码库中的 Symbol 和依赖分析
- 其他

谷歌公开的论文 *Dremel: Interactive Analysis of WebScaleDatasets*，总体介绍了 Dremel 的设计原理。该论文写于 2006 年，公开于 2010 年，Dremel 为了支持 Nested Data，做了很多设计的优化和权衡。

Dremel 系统有以下几个主要技术特点。

Dremel 是一个大规模的高并发系统。举例来说,磁盘的顺序读速度在 100MB/s 上下,那么在 1s 内处理 1TB 数据,意味着至少需要有 1 万个磁盘并发读,对于如此大量的读写,需要复杂的容错设计,少量节点的读失败(或慢操作)不能影响整体操作。

Dremel 支持嵌套的数据结构。互联网数据常常是非关系型的。Dremel 还支持灵活的数据模型,一种嵌套的(Nested)数据模型,类似于 Protocol Buffer 定义的数据结构。Dremel 采用列式方法存储这些数据,由于嵌套数据结构,Dremel 引入了一种树状的列式存储结构,方便嵌套数据查询。论文详细解释了嵌套数据类型的列存储,这个特性是 Druid 所缺少的,实现也是非常复杂的。

Dremel 采用层级的执行引擎。Dremel 在执行过程中,SQL 查询输入会转化成执行计划,并发处理数据。和 MapReduce 一样,Dremel 也需要和数据运行在一起,将计算移动到数据上面。所以它需要 GFS 这样的文件系统作为存储层。在设计之初,Dremel 并非 MapReduce 的替代品,它只是可以执行非常快的分析,在使用的时候,常常用它来处理 MapReduce 的结果集或者建立分析原型。

在使用 Dremel 时,工程师需要通过 MapReduce 将数据导入 Dremel,可以通过 MapReduce 的定时任务完成导入。在数据的实时性方面,论文中并没有讨论太多。

5. Apache Drill

Apache Drill 通过开源方式实现了谷歌 Dremel。Drill 架构的整个思想还是通过优化查询引擎,进行快速全表扫描,以快速返回结果,其高层架构示意图如图 2-15 所示。

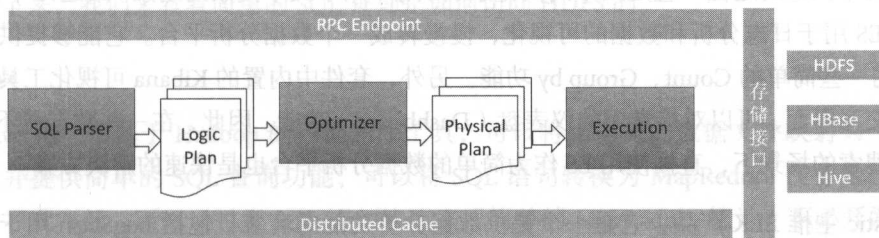


图 2-15 Apache Drill 高层架构示意图

Apache Drill 基于 SQL 的数据分析和商业智能(BI)引入了 JSON 文件模型,这使得用户能查询固定架构,支持各种格式和数据存储中的模式无关(schema-free)数据。该体系架构中的关系查询引擎和数据库构建是有先决条件的,即假设所有数据都有一个简单的静态架构。

Apache Drill 的架构是独一无二的。它是唯一一个支持复杂和无模式数据的柱状执行引擎（columnar execution engine），也是唯一一个能在查询执行期间进行数据驱动查询（和重新编译，也称为 schema discovery）的执行引擎。这些独一无二的性能使得 Apache Drill 在 JSON 文件模式下能实现记录断点性能（record-breaking performance）。该项目将会创建出开源版本的谷歌 Dremel Hadoop 工具（谷歌使用该工具为 Hadoop 数据分析工具的互联网应用提速），而“Drill”将有助于 Hadoop 用户实现更快查询海量数据集的目的。

目前，Drill 已经完成的需求和架构设计，总共分为以下 4 个组件。

Query language: 类似谷歌 BigQuery 的查询语言，支持嵌套模型，名为 DrQL。

Low-latency distribute execution engine: 执行引擎，可以支持大规模扩展和容错，并运行在上万台机器上计算数以 PB 的数据。

Nested data format: 嵌套数据模型，和 Dremel 类似，也支持 CSV、JSON、YAML 之类的模型，这样执行引擎就可以支持更多的数据类型。

Scalable data source: 支持多种数据源，现阶段以 Hadoop 为数据源。

6. Elasticsearch

Elasticsearch（ES）是 Elastic 公司推出的一个基于 Lucene 的分布式搜索服务系统，它是一个高可靠、可扩展、分布式的全文搜索引擎，提供了方便的 RESTful Web 接口。ES 采用 Java 语言开发，并作为 Apache 许可条款下的开放源码发布，它是流行的企业搜索引擎。与之类似的软件还有 Solr，两个软件有一定的相似性。

前几年，ES 的定位一直是文本的倒排索引引擎，用于文本搜索的场景。最近几年，Elastic 公司将 ES 用于日志分析和数据的可视化，慢慢转成一个数据分析平台。它能够提供类似于 OLAP 的一些简单的 Count、Group by 功能。另外，套件中内置的 Kibana 可视化工具提供了出色的交互界面，可以对接常用的仪表盘（Dashboard）功能。因此，在一些数据量不大，需要文本搜索的场景下，直接使用 ES 作为简单的数据分析平台也是快速的解决方案。

Elastic 主推 ELK 产品，它是一个提供数据分析功能的套装，包括 LogStash 用于数据收集；ES 用于数据索引；Kibana 用于可视化表现。

ES 内部使用了 Lucene 的倒排索引，每个 Term 后面都关联了相关的文档 ID 列表，这种结构比较适合基数较大的列，比如人名、单词等。ES 支持灵活的数据输入，支持无固定格式（schema-free）的数据输入，随时增加索引。

相比 Druid, ES 对于基数大的列能够提供完美的索引方案, 例如文本。ES 也提供了实时的数据摄入功能, 但是性能比 Druid 要慢很多, 因为它的索引过程更加复杂。另外一个显著不同是, ES 是 schema-free 的, 也就是说, 无须定义 Schema, 就可以直接插入 JSON 数据, 进行索引, 而且数据结构也支持数组等灵活的数据类型。Druid 需要定义清楚维度和指标列。还有一个很大区别, 就是 ES 会保持元素的文档数据, 而 Druid 在按照时间粒度数据聚合后, 原始数据将会丢弃, 因此无法召回具体的某一数据行。

最近几年, ES 一直在增加数据分析的能力, 包括各种聚合查询等, 性能提升也很快。在数据规模不大的情况下, ES 也是非常不错的选择。Druid 更善于处理更大规模、实时性更强的数据。

2.3.5 SQL on Hadoop/Spark

Hadoop 生态发展了多年, 越来越多的公司将重要的日志数据存入 Hadoop 的 HDFS 系统中, 数据的持久化和可靠性得到了保证, 但是如何快速挖掘出其中的价值却是很多公司的痛点。常用的分析过程有以下几种。

- 数据从 HDFS 导入 RDBMS/NoSQL。
- 基于 HDFS, 写代码通过 MapReduce 进行数据分析。
- 基于 HDFS, 编写 SQL 直接访问。
 - ◊ SQL 访问内部转换为 MapReduce 任务。
 - ◊ 访问引擎直接访问 HDFS 文件系统。

接下来, 我们来看看简单的 SQL 查询是如何访问 HDFS 的。

1. Hive

Hive 是一个基于 Hadoop 的数据仓库工具, 可以将结构化的数据文件映射为一张数据库表, 并提供简单的 SQL 查询功能, 可以将 SQL 语句转换为 MapReduce 任务运行。其优点是学习成本低, 可以通过类 SQL 语句快速实现简单的 MapReduce 统计, 不必开发专门的 MapReduce 应用, 十分适合数据仓库的统计分析。Hive 并不适合那些需要低延迟的应用, 例如联机事务处理 (OLTP)。Hive 查询操作过程严格遵守 Hadoop MapReduce 的作业执行模型, 整个查询过程也比较慢, 不适合实时数据分析。

几乎所有的 Hadoop 环境都会配置 Hive 的应用, 虽然 Hive 易用, 但内部的 MapReduce 操作还是会带来非常慢的查询体验。所有尝试 Hive 的公司, 几乎都会转型到 Impala 的应用。

2. Impala

Impala 是 Cloudera 受谷歌 Dremel 启发开发的实时交互 SQL 大数据查询工具，使用 C++ 编写，通过使用与商用 MPP 类似的分布式查询引擎（由 Query Planner、Query Coordinator 和 Query Exec Engine 三部分组成），可以直接从 HDFS 或 HBase 中用 SELECT、JOIN 和统计函数查询数据，从而大大降低了延迟。

Impala 使用的列存储格式是 Parquet。Parquet 实现了 Dremel 中的列存储，未来还将支持 Hive 并添加字典编码、游程编码等功能。在 Cloudera 的测试中，Impala 的查询效率比 Hive 有数量级的提升，因为 Impala 省去了 MapReduce 的过程，减少了中间结果落盘的问题。Impala 的数据流程图如图 2-16 所示。

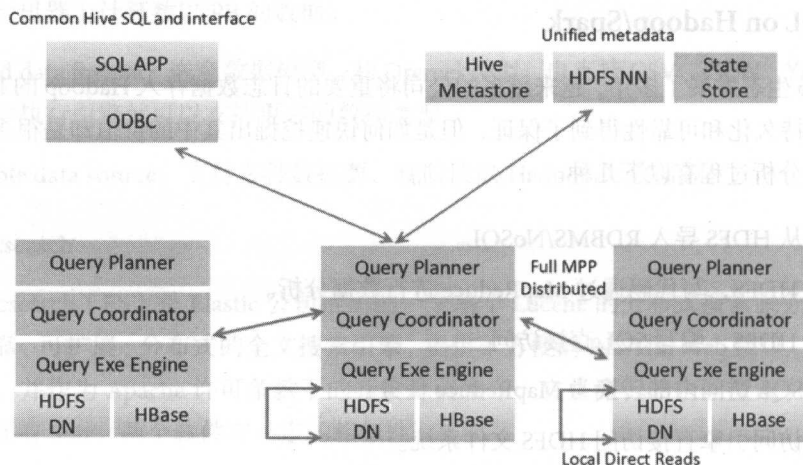


图 2-16 Impala 的数据流程图

相比 Druid，Impala 需要将数据格式转换成 Parquet 格式才能进行查询，有时候数据转换也会花费不少时间。

3. Facebook Presto

Presto 出身名门，来自于 Facebook，从出生起就受到关注。其主要采用 Java 编写。Presto 是一个分布式 SQL 查询引擎，它被设计专门用于进行高速、实时的数据分析。它支持标准的 ANSI SQL，包括复杂查询、聚合（Aggregation）、连接（Join）和窗口函数（Window function）。图 2-17 展示了简化的 Presto 系统架构示意图。

Presto 的运行模型和 Hive 或 MapReduce 有着本质的区别。Hive 将查询翻译成多阶段的 MapReduce 任务，一个接着一个运行，每一个任务从磁盘读取输入数据并且将中间结果输出

到磁盘上。然而，Presto 引擎没有使用 MapReduce，它使用了一个定制的查询和执行引擎及响应的操作符来支持 SQL 语法。除了改进的调度算法之外，所有的数据处理都是在内存中进行的。通过软件的优化，形成处理的流水线，以避免不必要的磁盘读写和额外的延迟。这种流水线式的执行模型会在同一时间运行多个数据处理段，一旦数据可用就会将数据从一个处理段传入下一个处理段。这样的方式会大大减少各种查询的端到端响应时间。

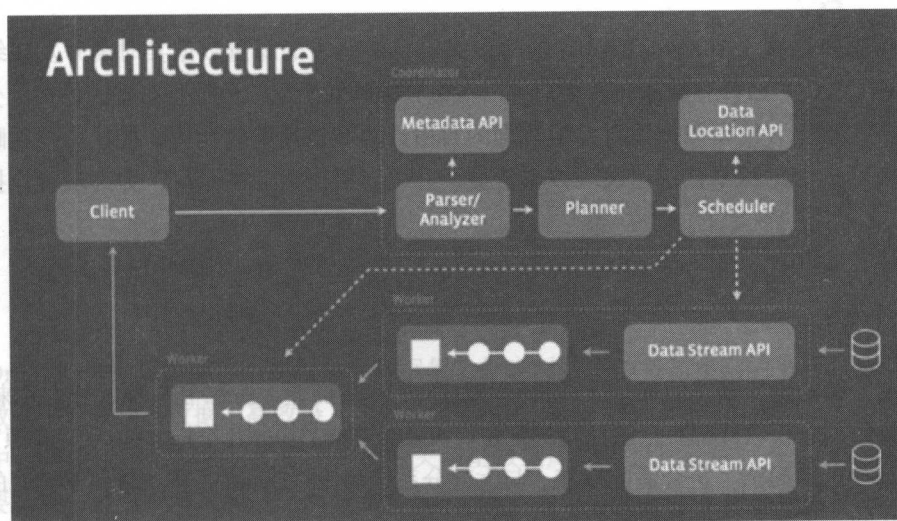


图 2-17 简化的 Presto 系统架构示意图

相比 Druid，Presto 主要是解决 SQL 查询引擎的问题，将 SQL 查询转换成分布式任务，快速到数据存储区获得必要的数 据，并且返回结果。它并不直接去优化存储结构，而是直接访问 HDFS 或者其他数据存储层。

2.3.6 数据分析云服务

1. Redshift

Amazon Redshift 是一种快速的、完全托管的 PB 级别数据仓库，可以方便用户使用现有的商业智能工具，以一种经济的方式轻松分析所有数据。Amazon Redshift 使用列存储技术改善 I/O 效率并跨越多个节点平行放置查询，从而提供快速的查询性能。Amazon Redshift 提供了定制的 JDBC 和 ODBC 驱动程序，可以从控制台的“连接客户端”选项卡中进行下载，以使用各种大量熟悉的 SQL 客户端。当然也可以使用标准的 PostgreSQL JDBC 和 ODBC 驱动程序。数据加载速度与集群大小，与 Amazon S3、Amazon DynamoDB、Amazon Elastic MapReduce、Amazon Kinesis 或任何启用 SSH 的主机的集成呈线性扩展关系。

相比 Druid, Redshift 是一种 SaaS 服务。Redshift 内部使用了亚马逊已经取得授权的 ParAccel 技术。ParAccel 是一家专注提供数据分析服务的老牌技术公司,曾推出自己的列式存储的数据仓库产品。Druid 适合分析大数据量的流式数据,也能够实时加载和聚合数据。Druid 用索引来提高带过滤查询的速度,虽然索引结构简单,但是效率很高。

2. 阿里云数据仓库服务

分析型数据库 (Analytic DB), 是阿里巴巴自主研发的海量数据实时高并发在线分析 (Realtime OLAP) 云计算服务,使用户可以在毫秒级针对千亿级数据进行即时的多维分析透视和业务探索。分析型数据库对海量数据的自由计算和极速响应能力,能让用户在瞬息之间进行灵活的数据探索,快速发现数据价值,并可直接嵌入业务系统,为终端客户提供分析服务。

2.4 小结

数据分析的世界繁花似锦,虽然我们可以通过开源/商业、SaaS/私有部署等方式来分类,但是每种数据分析软件都有自己独特的定位。如果需要给 Druid 几个标签的话,“开源”、“实时”、“高效”、“简洁”是合适的标签。与大部分系统相比,Druid 系统功能属于精简的,性能是出众的,实时支持也是超群的。

参考资料

<http://www.yankay.com/google-dremel-rationale/>

<http://www.teradatawiki.net/2013/09/Teradata-Architecture.html>

<http://opentsdb.net/img/tsdb-architecture.png>

第3章

架构详解

Druid 的目标是提供一个能够在大数据集上做实时数据消费与探索的平台。然而，对普遍的数据平台来说，数据的高效摄入与快速查询往往是一对难以两全的指标，因此常常需要在其中做一些取舍与权衡。比如，传统的关系型数据库如果想在查询时有更快的响应速度，就需要牺牲一些数据写入的性能以完成索引的创建；反之，如果想获得更快的写入速度，往往要放弃一些索引的创建，就势必在查询的时候付出更高的性能代价。相比之下，Druid 却能够同时提供性能卓越的数据实时摄入与复杂的查询性能。它是怎么做到的呢？答案是通过其独到的架构设计、基于 DataSource 与 Segment 的数据结构，以及在许多系统细节上的优秀设计与实现。本章将着重介绍 Druid 的架构设计与数据结构，其他出色的细节设计与实现将在第 8 章“核心源代码探析”中详细介绍。

3.1 Druid 架构概览

关于 Druid 架构，我们先通过其总体架构图来做一个概要了解，如图 3-1 所示。

Druid 总体架构图显示出 Druid 自身包含以下 4 类节点。

- 实时节点 (Realtime Node)：即时摄入实时数据，以及生成 Segment 数据文件。
- 历史节点 (Historical Node)：加载已生成好的数据文件，以供数据查询。
- 查询节点 (Broker Node)：对外提供数据查询服务，并同时从实时节点与历史节点查询数据，合并后返回给调用方。
- 协调节点 (Coordinator Node)：负责历史节点的数据负载均衡，以及通过规则 (Rule)

管理数据的生命周期。

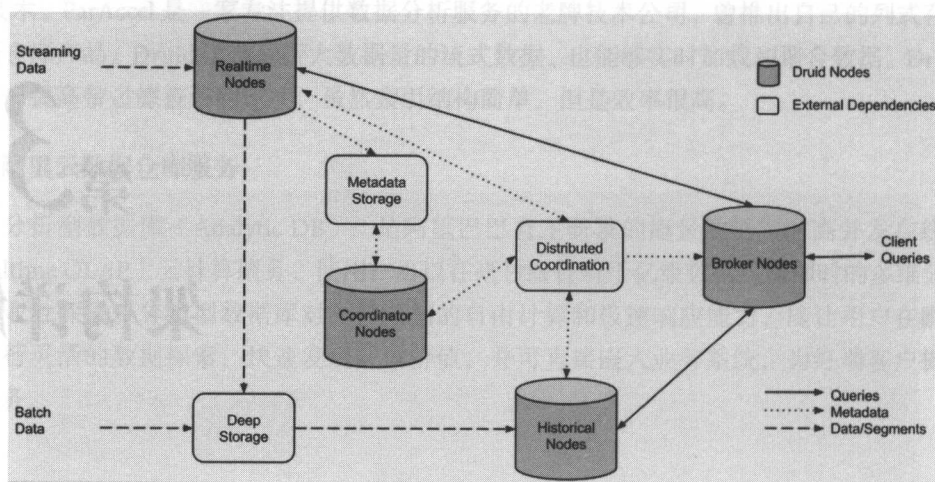


图 3-1 Druid 总体架构图

同时，集群还包含以下三类外部依赖。

- 元数据库 (Metastore): 存储 Druid 集群的原数据信息，比如 Segment 的相关信息，一般用 MySQL 或 PostgreSQL。
- 分布式协调服务 (Coordination): 为 Druid 集群提供一致性协调服务的组件，通常为 Zookeeper。
- 数据文件存储库 (DeepStorage): 存放生成的 Segment 数据文件，并供历史节点下载。对于单节点集群可以是本地磁盘，而对于分布式集群一般是 HDFS 或 NFS。

从数据流转的角度来看，数据从架构图的左侧进入系统，分为实时流数据与批量数据。实时流数据会被实时节点消费，然后实时节点将生成的 Segment 数据文件上传到数据文件存储库；而批量数据经过 Druid 集群消费后（具体方法后面的章节会进行介绍）会被直接上传到数据文件存储库。同时，查询节点会响应外部的查询请求，并将分别从实时节点与历史节点查询到的结果合并后返回。

3.2 Druid 架构设计思想

对于目前大多数 Druid 的使用场景来说，Druid 本质上是一个分布式的时序数据库，而对于一个数据库的性能来说，其数据的组织方式至关重要。为了更好地阐述 Druid 的架构设计思想，我们得先从数据库的文件组织方式聊起。

众所周知，数据库的数据大多存储在磁盘上，而磁盘的访问相对内存的访问来说是一项很耗时的操作，对比如图 3-2 所示。因此，提高数据库数据的查找速度的关键点之一便是尽量减少磁盘的访问次数。

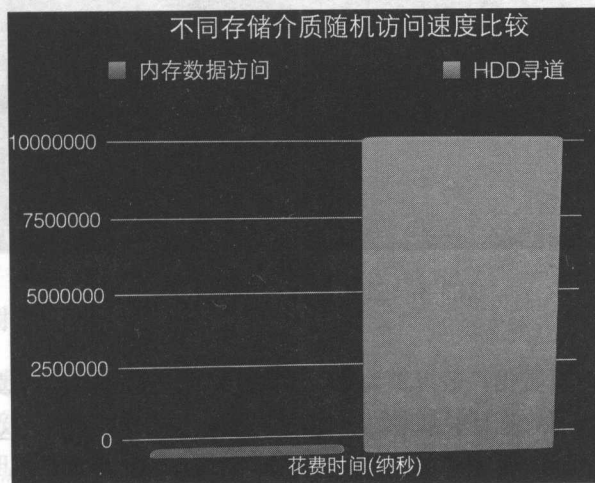


图 3-2 磁盘与内存的访问速度对比

为了加速数据库数据的访问，大多传统的关系型数据库都会使用特殊的数据结构来帮助查找数据，这种数据结构叫作索引（Index）。对于传统的关系型数据库，考虑到经常需要范围查找某一批数据，因此其索引一般不使用 Hash 算法，而使用树（Tree）结构。然而，树结构的种类很多，却不一定都适合用于做数据库索引。

3.2.1 索引对树结构的选择

1. 二叉查找树与平衡二叉树

最常见的树结构是二叉查找树（Binary Search Tree），它就是一棵二叉有序树：保证左子树上所有节点的值都小于根节点的值，而右子树上所有节点的值都大于根节点的值。其优点在于实现简单，并且树在平衡的状态下查找效率能达到 $O(\log_2 N)$ ；缺点是在极端非平衡情况下查找效率会退化到 $O(N)$ ，因此很难保证索引的效率。二叉查找树的查找效率如图 3-3 所示。

针对上述二叉查找树的缺点，人们很自然就想到是否能用平衡二叉树（Balanced Binary Tree）来解决这个问题。但是平衡二叉树依然有个比较大的问题：它的树高为 $\log_2 N$ ——对于索引树来说，树的高度越高，意味着查找所要花费的访问次数越多，查询效率越低。

了日志结构方法的优点,又通过将数据文件预排序克服了日志结构方法随机读性能较差的问题。尽管当时 LSM-tree 新颖且优势鲜明,但它真正声名鹊起却是在 10 年之后的 2006 年,那年谷歌的一篇使用了 LSM-tree 技术的论文 *Bigtable: A Distributed Storage System for Structured Data* 横空出世,在分布式数据处理领域掀起了一阵旋风,随后两个声名赫赫的大数据开源组件(2007 年的 HBase 与 2008 年的 Cassandra,目前两者同为 Apache 顶级项目)直接在其思想基础上破茧而出,彻底改变了大数据基础组件的格局,同时也极大地推广了 LSM-tree 技术。

LSM-tree 最大的特点是同时使用了两部分分类树的数据结构来存储数据,并同时提供查询。其中一部分数据结构(C_0 树)存在于内存缓存(通常叫作 memtable)中,负责接受新的数据插入更新以及读请求,并直接在内存中对数据进行排序;另一部分数据结构(C_1 树)存在于硬盘上(这部分通常叫作 sstable),它们是由存在于内存缓存中的 C_0 树冲写到磁盘而成的,主要负责提供读操作,特点是有序且不可被更改。LSM-tree 的 C_0 与 C_1 部分如图 3-6 所示。

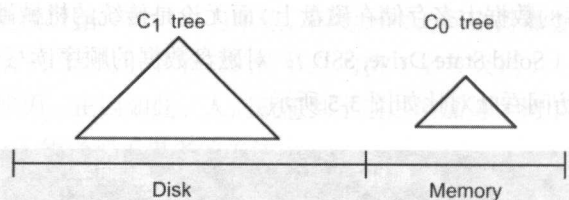


图 3-6 LSM-tree 的 C_0 与 C_1 部分

LSM-tree 的另一大特点是除了使用两部分分类树的数据结构外,还会使用日志文件(通常叫作 commit log)来为数据恢复做保障。这三类数据结构的协作顺序一般是:所有的新插入与更新操作都首先被记录到 commit log 中——该操作叫作 WAL (Write Ahead Log),然后再写到 memtable,最后当达到一定条件时数据会从 memtable 冲写到 sstable,并抛弃相关的 log 数据;memtable 与 sstable 可同时供查询;当 memtable 出问题时,可从 commit log 与 sstable 中将 memtable 的数据恢复。

我们可以参考 HBase 的架构来体会其架构中基于 LSM-tree 的部分特点。按照 WAL 的原则,数据首先会写到 HBase 的 HLog(相当于 commit log)里,然后再写到 MemStore(相当于 memtable)里,最后会冲写到磁盘 StoreFile(相当于 sstable)中。这样 HBase 的 HRegionServer 就通过 LSM-tree 实现了数据文件的生成。HBase LSM-tree 架构示意图如图 3-7 所示。

LSM-tree 的这种结构非常有利于数据的快速写入(理论上可以接近磁盘顺序写速度),但是不利于读——因为理论上读的时候可能需要同时从 memtable 和所有硬盘上的 sstable 中查询数据,这样显然会对性能造成较大的影响。为了解决这个问题,LSM-tree 采取了以下主要的相关措施。

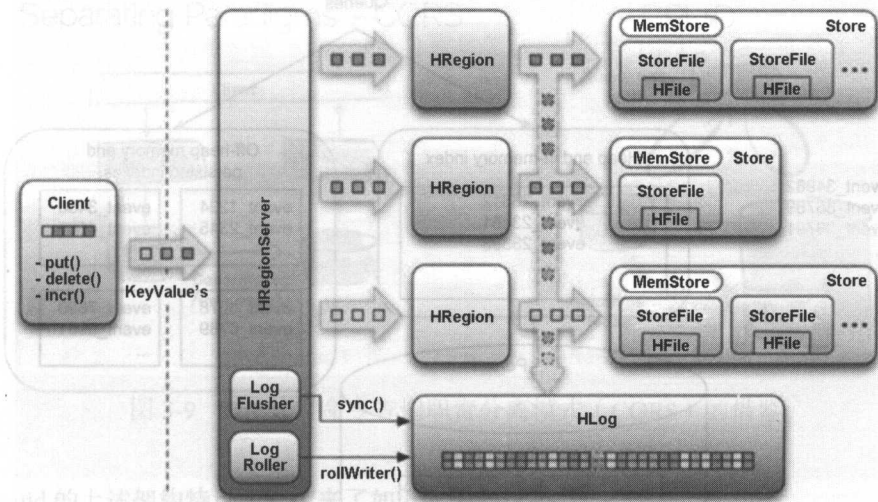


图 3-7 HBase LSM-tree 架构示意图

- 定期将硬盘上小的 sstable 合并（通常叫作 Merge 或 Compaction 操作）成大的 sstable，以减少 sstable 的数量。而且，平时的数据更新删除操作并不会更新原有的数据文件，只会将更新删除操作加到当前的数据文件末端，只有在 sstable 合并的时候才会真正将重复的操作或更新去重、合并。
- 对每个 sstable 使用布隆过滤器（Bloom Filter），以加速对数据在该 sstable 的存在性进行判定，从而减少数据的总查询时间。

3.2.2 Druid 总体架构

LSM-tree 显然比较适合那些数据插入操作远多于数据更新删除操作与读操作的场景，同时 Druid 在一开始就是为时序数据场景设计的，而该场景正好符合 LSM-tree 的优势特点，因此 Druid 架构便顺理成章地吸取了 LSM-tree 的思想。

Druid 的类 LSM-tree 架构中的实时节点（Realtime Node）负责消费实时数据，与经典 LSM-tree 架构不同的是，Druid 不提供日志及实行 WAL 原则，实时数据首先会被直接加载进实时节点内存中的堆结构缓存区（相当于 memtable），当条件满足时，缓存区里的数据会被冲写到硬盘上形成一个数据块（Segment Split），同时实时节点又会立即将新生成的数据块加载到内存中的非堆区，因此无论是堆结构缓存区还是非堆区里的数据，都能够被查询节点（Broker Node）查询。实时节点数据块的生成示意图如图 3-8 所示。

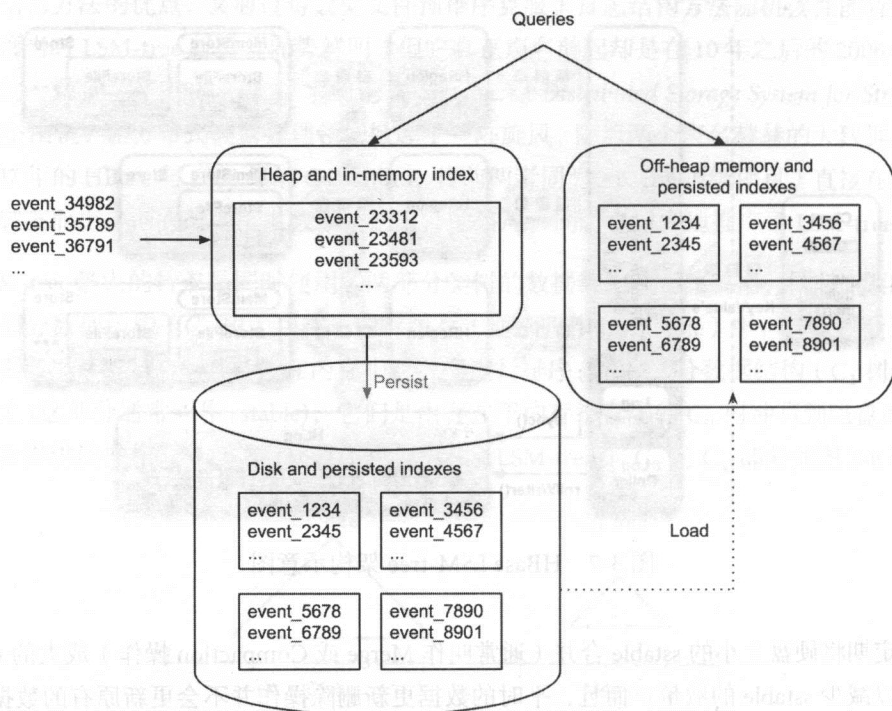


图 3-8 实时节点数据块的生成示意图

同时，实时节点会周期性地 将磁盘上同一个时间段内生成的所有数据块合并为一个大的数据块（Segment）。这个过程在实时节点中叫作 Segment Merge 操作，也相当于 LSM-tree 架构中的数据合并操作（Compaction）。合并好的 Segment 会立即被实时节点上传到数据文件存储库（DeepStorage）中，随后协调节点（Coordinator Node）会指导一个历史节点（Historical Node）去文件存储库，将新生成的 Segment 下载到其本地磁盘中。当历史节点成功加载到 Segment 后，会通过分布式协调服务（Coordination）在集群中声明其从此刻开始负责提供该 Segment 的查询，当实时节点收到该声明后也会立即向集群声明其不再提供该 Segment 的查询，接下来查询节点会转从该历史节点查询此 Segment 的数据。而对于全局数据来说，查询节点会同时从实时节点（少量当前数据）与历史节点（大量历史数据）分别查询，然后做一个结果的整合，最后再返回给用户。Druid 的这种架构安排实际上也在一定程度上借鉴了命令查询职责分离模式（Command Query Responsibility Segregation, CQRS）——这也是 Druid 不同于 HBase 等 LSM-tree 系架构的一个显著特点。Druid 对命令查询职责分离模式（CQRS）的借鉴如图 3-9 所示。

Separating Paradigms - CQRS

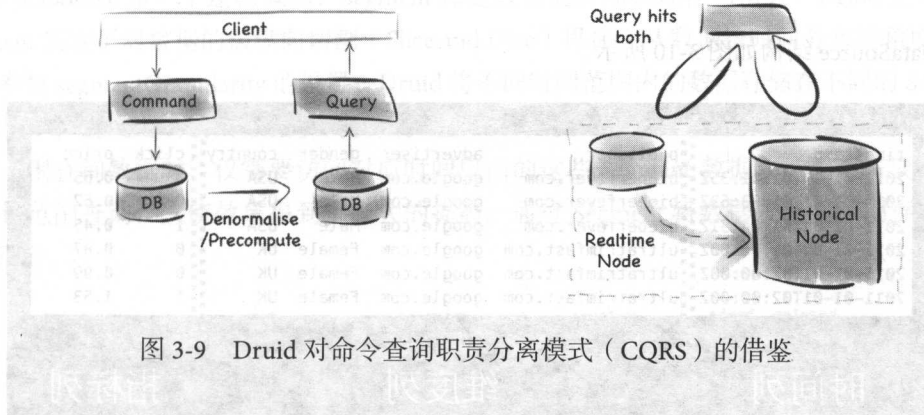


图 3-9 Druid 对命令查询职责分离模式（CQRS）的借鉴

Druid 的上述架构特点为其带来了如下显著的优势。

- 类 LSM-tree 架构使得 Druid 能够保证数据的高速写入，并且能够提供比较快速的实时查询，这十分符合许多时序数据的应用场景。
- 由于 Druid 在设计之初就不提供对已有数据的更改，以及不实现传统 LSM-tree 架构中普遍应用的 WAL 原则，虽然这样导致了 Druid 不适用于某些需要数据更新的场景，也降低了数据完整性的保障，但 Druid 相对其他传统的 LSM-tree 架构实现来说也着实减少了不少数据处理的工作量，因此让自己在性能方面更胜一筹。
- Druid 对命令查询职责分离模式的借鉴也使得自己的组件职责分明、结构更加清晰明了，方便针对不同模块进行针对性的优化。

3.2.3 基于 DataSource 与 Segment 的数据结构

与 Druid 架构相辅相成的是其基于 DataSource 与 Segment 的数据结构，它们共同成就了 Druid 的高性能优势。

1. DataSource 结构

若与传统的关系型数据库管理系统（RDBMS）做比较，Druid 的 DataSource 可以理解为 RDBMS 中的表（Table）。正如前面章节中所介绍的，DataSource 的结构包含以下几个方面。

- 时间列（TimeStamp）：表明每行数据的时间值，默认使用 UTC 时间格式且精确到毫秒级别。这个列是数据聚合与范围查询的重要维度。
- 维度列（Dimension）：维度来自于 OLAP 的概念，用来标识数据行的各个类别信息。

- 指标列 (Metric): 指标对应于 OLAP 概念中的 Fact, 是用于聚合和计算的列。这些指标列通常是一些数字, 计算操作通常包括 Count、Sum 和 Mean 等。

DataSource 结构如图 3-10 所示。

timestamp	publisher	advertiser	gender	country	click	price
2011-01-01T01:01:35Z	bieberfever.com	google.com	Male	USA	0	0.65
2011-01-01T01:03:63Z	bieberfever.com	google.com	Male	USA	0	0.62
2011-01-01T01:04:51Z	bieberfever.com	google.com	Male	USA	1	0.45
2011-01-01T01:00:00Z	ultratrimfast.com	google.com	Female	UK	0	0.87
2011-01-01T02:00:00Z	ultratrimfast.com	google.com	Female	UK	0	0.99
2011-01-01T02:00:00Z	ultratrimfast.com	google.com	Female	UK	1	1.53

时间列

维度列

指标列

图 3-10 DataSource 结构

无论是实时数据消费还是批量数据处理, Druid 在基于 DataSource 结构存储数据时即可选择对任意的指标列进行聚合 (Roll Up) 操作。该聚合操作主要基于维度列与时间范围两方面的情况。

- 同维度列的值做聚合: 所有维度列的值都相同时, 这一类行数据符合聚合操作, 比如对于所有维度组合 “publisher advertiser gender country” 维度值同为 “ultratrimfast.com google.com Male USA” 或同为 “bieberfever.com google.com Male USA” 的行。
- 对指定时间粒度内的值做聚合: 符合参数 queryGranularity 指定的范围, 比如时间列的值为同 1 分钟内的所有行, 聚合操作相当于对数据表所有列做了 Group By 操作, 比如 “GROUP BY timestamp, publisher, advertiser, gender, country :: impressions = COUNT(1), clicks = SUM(click), revenue = SUM(price) ”。图 3-11 显示的是执行聚合操作后 DataSource 的数据情况。

timestamp	publisher	advertiser	gender	country	impressions	clicks	revenue
2011-01-01T01:00:00Z	ultratrimfast.com	google.com	Male	USA	1800	25	15.70
2011-01-01T01:00:00Z	bieberfever.com	google.com	Male	USA	2912	42	29.18
2011-01-01T02:00:00Z	ultratrimfast.com	google.com	Male	UK	1953	17	17.31
2011-01-01T02:00:00Z	bieberfever.com	google.com	Male	UK	3194	170	34.01

图 3-11 DataSource 聚合后的数据情况

相对于其他时序数据库, Druid 在数据存储时便可对数据进行聚合操作是其一大特点, 该特点使得 Druid 不仅能够节省存储空间, 而且能够提高聚合查询的效率。

2. Segment 结构

DataSource 是一个逻辑概念，Segment 却是数据的实际物理存储格式，Druid 正是通过 Segment 实现了对数据的横纵向切割（Slice and Dice）操作。从数据按时间分布的角度来看，通过参数 segmentGranularity 的设置，Druid 将不同时间范围内的数据存储在不同的 Segment 数据块中，这便是所谓的数据横向切割。这种设计为 Druid 带来一个显而易见的优点：按时间范围查询数据时，仅需要访问对应时间段内的这些 Segment 数据块，而不需要进行全表数据范围查询，这使效率得到了极大的提高。通过 Segment 将数据按时间范围存储如图 3-12 所示。

timestamp	page	language	city	country	...	added	deleted
2011-01-01T00:01:35Z	Justin Bieber	en	SF	USA		10	65
2011-01-01T00:03:63Z	Justin Bieber	en	SF	USA		15	62
2011-01-01T00:04:51Z	Justin Bieber	en	SF	USA		32	45
Segment 2011-01-01T00/2011-01-01T01							
2011-01-01T01:00:00Z	Ke\$ha	en	Calgary	CA		17	87
Segment 2011-01-01T01/2011-01-01T02							
2011-01-01T02:00:00Z	Ke\$ha	en	Calgary	CA		43	99
2011-01-01T02:00:00Z	Ke\$ha	en	Calgary	CA		12	53
Segment 2011-01-01T02/2011-01-01T03							

图 3-12 通过 Segment 将数据按时间范围存储

同时，在 Segment 中也面向列进行数据压缩存储，这便是所谓的数据纵向切割。而且在 Segment 中使用了 Bitmap 等技术对数据的访问进行了优化。

3.3 扩展系统

Druid 在设计之初就希望自己的系统功能能够比较轻松地被扩展，特别是对于有特殊需求的用户，因此它通过实现了一个扩展系统（Extension System），让很多组件功能能够通过扩展（Extension）的形式轻易地在 Druid 平台上进行加载与更换。

3.3.1 主要的扩展

目前 Druid 项目自带的一些扩展如下。

名称	描述
druid-avro-extensions	使得 Druid 能够摄入 avro 格式的数据。一般可在 dataSchema 中的 parser 部分进行指定使用

续表

名称	描述
druid-hdfs-storage	使用 HDFS 作为 DeepStorage
druid-s3-extensions	使用 AWS S3 作为 DeepStorage
druid-histogram	对 histograms 值进行近似计算
druid-datasketches	使得 Druid 的聚合操作能够使用 datasketches 库
druid-namespace-lookup	命名空间发现
druid-kafka-eight	消费 Kafka 数据的 Firehose（使用 Kafka High Level Consumer API）
druid-kafka-extraction-namespace	基于 Kafka 的命名空间发现，依赖于命名空间发现扩展
mysql-metadata-storage	使用 MySQL 作为元数据数据库
postgresql-metadata-storage	使用 PostgreSQL 作为元数据数据库

目前 Druid 社区贡献的一些扩展如下。

名称	描述
druid-azure-extensions	使用 Microsoft Azure 作为 DeepStorage
druid-cassandra-storage	使用 Cassandra 作为 DeepStorage
druid-cloudfiles-extensions	基于 Rackspace Cloudfiles 的 DeepStorage 与 Firehose
druid-kafka-eight-simpleConsumer	消费 Kafka 数据的 Firehose（使用 Kafka Low Level Consumer API）
druid-rabbitmq	基于 rabbitmq 的 Firehose
graphite-emitter	Graphite metrics 的发射器（emitter）

3.3.2 下载与加载扩展

通过 Druid 自身提供的 pull-deps 工具下载依赖到本地仓库。比如想要利用 pull-deps 下载 druid-rabbitmq, mysql-metadata-storage 与 hadoop-client（版本 2.3.0 和 2.4.0）到本地，可使用如下命令。

```
java -classpath "/my/druid/library/*" io.druid.cli.Main tools pull-deps --clean -c io.druid.extensions:mysql-metadata-storage:0.9.0 -c io.druid.extensions.contrib:druid-
```

```
rabbitmq:0.9.0 -h org.apache.hadoop:hadoop-client:2.3.0 -h org.apache.hadoop:hadoop-client:2.4.0
```

加载扩展有以下两种方法。

- 将扩展加载到 Druid Service 的 classpath 中，Druid 便能加载到相关扩展。
- 在 common.runtime.properties 文件中通过 druid.extensions.directory 指定扩展目录。

3.4 实时节点

实时节点（Realtime Node）主要负责即时摄入实时数据，以及生成 Segment 数据文件，其独到的设计使其拥有超强的数据摄入速度。

3.4.1 Segment 数据文件的制造与传播

实时节点通过 Firehose 来消费实时数据。Firehose 是 Druid 中消费实时数据的模型，可以有不同的具体实现，比如 Druid 自带的基于 Kafka High Level API 实现的用于消费 Kafka 数据的 druid-kafka-eight Firehose，以及社区贡献的基于 Kafka Low Level API 实现的 druid-kafka-eight-simpleConsumer Firehose。同时，实时节点会通过另一个用于生成 Segment 数据文件的模块 Plumber（具体实现有 RealtimePlumber 等）按照指定的周期，按时将本周期内生产出的所有数据块合并成一个大的 Segment 数据文件。

Segment 数据文件从制造到传播要经历一个完整的流程，步骤如下。

- （1）实时节点生产出 Segment 数据文件，并将其上传到 DeepStorage 中。
- （2）Segment 数据文件的相关元数据信息被存放到 MetaStore（即 MySQL）里。
- （3）Master 节点（即 Coordinator 节点）从 MetaStore 里得知 Segment 数据文件的相关元数据信息后，将其根据规则的设置分配给符合条件的历史节点。
- （4）历史节点得到指令后会主动从 DeepStorage 中拉取指定的 Segment 数据文件，并通过 Zookeeper 向集群声明其负责提供该 Segment 数据文件的查询服务。
- （5）实时节点丢弃该 Segment 数据文件，并向集群声明其不再提供该 Segment 数据文件的查询服务。

Segment 数据文件的传播过程如图 3-13 所示。

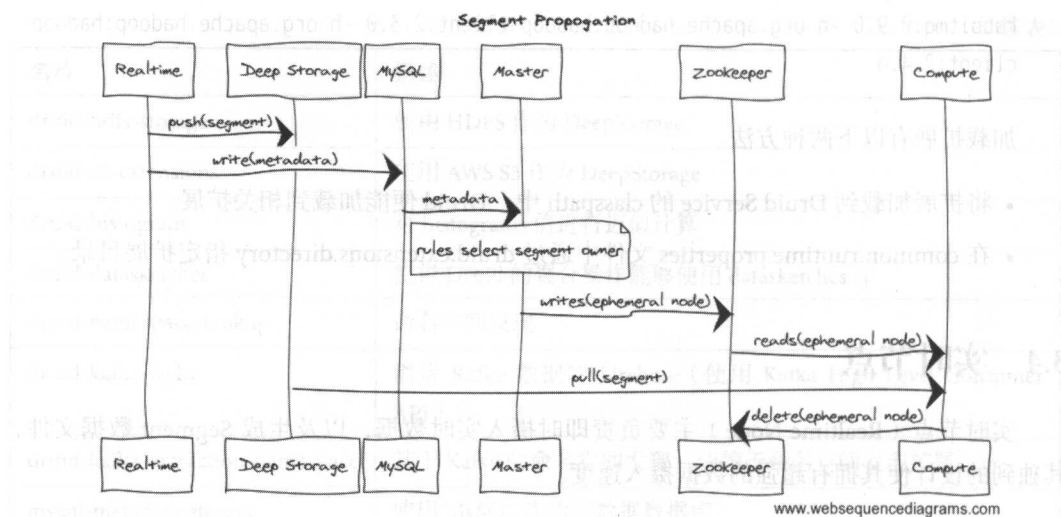


图 3-13 Segment 数据文件的传播过程

3.4.2 高可用性与可扩展性

从设计上看，实时节点拥有很好的可扩展性与高可用性。对于 Druid 0.7.3 版本来说，其消费 Kafka 数据时一般使用 Druid 自带的用于消费 Kafka 数据的 druid-kafka-eight Firehose，使用 pull 的方式从 Kafka 获取数据，而该 Firehose 能够让 Druid 有较好的可扩展性和高可用性。我们可以使用一组实时节点组成一个 Kafka Consumer Group 来共同消费同一个 Kafka Topic 的数据，各个节点会负责独立消费一个或多个该 Topic 所包含的 Partition 的数据，并且保证同一个 Partition 不会被多于一个的实时节点消费。当每一个实时节点完成部分数据的消费后，会主动将数据消费进度（Kafka Topic Offset）提交到 Zookeeper 集群。这样，当这个节点不可用时，该 Kafka Consumer Group 会立即在组内对所有可用节点进行 Partition 的重新分配，接着所有节点将会根据记录在 Zookeeper 集群里的每一个 Partition 的 Offset 来继续消费未曾被消费的数据，从而保证所有数据在任何时候都会被 Druid 集群至少消费一次，进而实现了这个角度上的高可用性。同样的道理，当集群中添加新的实时节点时也会触发相同的事件，从而保证了实时节点能够轻松实现线性扩展。

可惜的是，这个方法并不是毫无破绽的。因为，当一个 Kafka Consumer Group 内的实时节点不可用时，该方法虽然能够保证它所负责的 Partition 里未曾被消费的数据能被其他存活的实时节点分配并且被消费，但是该不可用实时节点上的已经被消费，尚未被传送到 Deep Storage 上且被其他历史节点下载的 Segment 数据却会被集群所遗漏，从而形成了这个基于 druid-kafka-eight Firehose 的消费方案的高可用性方面的一个缺陷。解决这个问题一般有两种方式。

- 想办法让不可用的实时节点重新回到集群成为可用节点，那么当它重启的时候会将其之前已经生成但尚未被上传的 Segment 数据文件统统都加载回来，并最终会将其合并并上传到 DeepStorage，保证了数据的完整性。
- 使用 Tranquility 与索引服务（Indexing Service）对具体的 Kafka Topic Partition 进行精确的消费与备份。由于 Tranquility 可以通过 push 的方式将指定的数据推向 Druid 集群，因此它可以同时对同一个 Partition 制造多个副本（replica）。所以当某个数据消费的任务失败时，系统依然可以准确地选择使用另外一个相同任务所创建的 Segment 数据块。

3.5 历史节点

历史节点（Historical Node）负责加载已生成好的数据文件以提供数据查询。由于 Druid 的数据文件有不可更改性，因此历史节点的工作就是专注于提供数据查询。

3.5.1 内存为王的查询之道

历史节点在启动的时候，首先会检查自己的本地缓存（Local Cache）中已存在的 Segment 数据文件，然后从 DeepStorage 中下载属于自己但目前不在自己本地磁盘上的 Segment 数据文件。无论是哪种查询，历史节点都会首先将相关 Segment 数据文件从磁盘加载到内存，然后再提供查询服务，如图 3-14 所示。

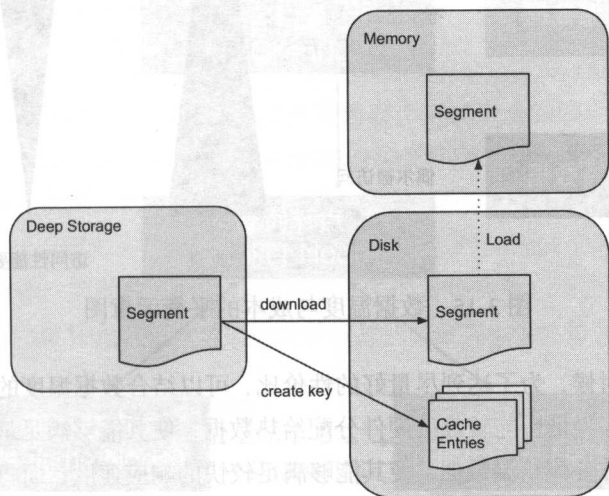


图 3-14 历史节点加载 Segment 数据文件

不难看出,历史节点的查询效率受内存空间富余程度的影响很大:内存空间富余,查询时需要从磁盘加载数据的次数就少,查询速度就快;反之,查询时需要从磁盘加载数据的次数就多,查询速度就相对较慢。因此,原则上历史节点的查询速度与其内存空间大小和所负责的 Segment 数据文件大小之比成正比关系。

3.5.2 层的分组功能

对于分布式系统来说,在规划存储时需要同时考虑硬件的异构性与数据温度,并在它们之间找到一个满足实际需求的平衡点。所谓硬件的异构性是指在集群中不同节点的硬件性能不同;而数据温度则是用来形容数据被访问的频繁程度——越频繁温度越高。根据数据温度,数据可被笼统地分为以下三类。

- 热数据:经常被访问。特点是总的的数据量不算大,但要求响应速度最快。
- 温数据:不经常被访问。特点是总的的数据量一般,且要求响应速度尽量快。
- 冷数据:偶尔被访问。特点是总的的数据量占比最大,但响应速度不用很快。

数据温度与成本的平衡示意图如图 3-15 所示。

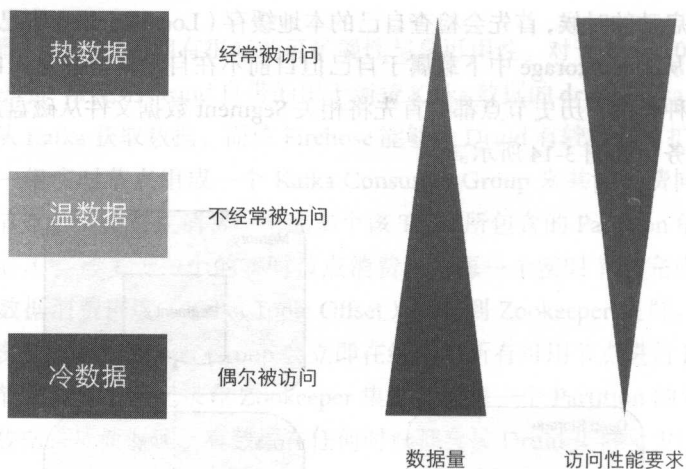


图 3-15 数据温度与成本的平衡示意图

在规划存储的时候,为了达到尽量好的性价比,可以结合数据温度的特点与硬件的优劣程度来综合考虑:将少量性能较好的硬件分配给热数据,使其能够满足最快的响应速度;将更多性能中等的硬件分配给温数据,使其能够满足较快的响应速度;而将最多的性能普通的硬件分配给冷数据,使其能够满足较大的存储空间要求。基于上述的理论考虑,在实践中往往会要求将硬件资源根据性能容量等指标分为不同的组,并为这些硬件都打上组标签。

Druid 也考虑到了这个问题, 因此提出了层 (Tier) 的概念: 将集群中所有的历史节点根据性能容量等指标分为不同的层, 并且可以让不同性质的 DataSource 使用不同的层来存储 Segment 数据文件, 以达到效率与成本相对平衡的状态。

3.5.3 高可用性与可扩展性

历史节点拥有极佳的可扩展性与高可用性。新的历史节点被添加后, 会通过 Zookeeper 被协调节点发现, 然后协调节点将会自动分配相关的 Segment 给它; 原有的历史节点被移出集群后, 同样会被协调节点发现, 然后协调节点会将原本分配给它的 Segment 重新分配给其他处于工作状态的历史节点。

3.6 查询节点

查询节点 (Broker Node) 对外提供数据查询服务, 并同时从实时节点与历史节点查询数据, 合并后返回给调用方。

3.6.1 查询中枢点

在常规情况下, Druid 集群直接对外提供查询的节点只有查询节点, 而查询节点会将从实时节点与历史节点查询到的数据合并后返回给客户端。因此, 查询节点便是整个集群的查询中枢点, 如图 3-16 所示。

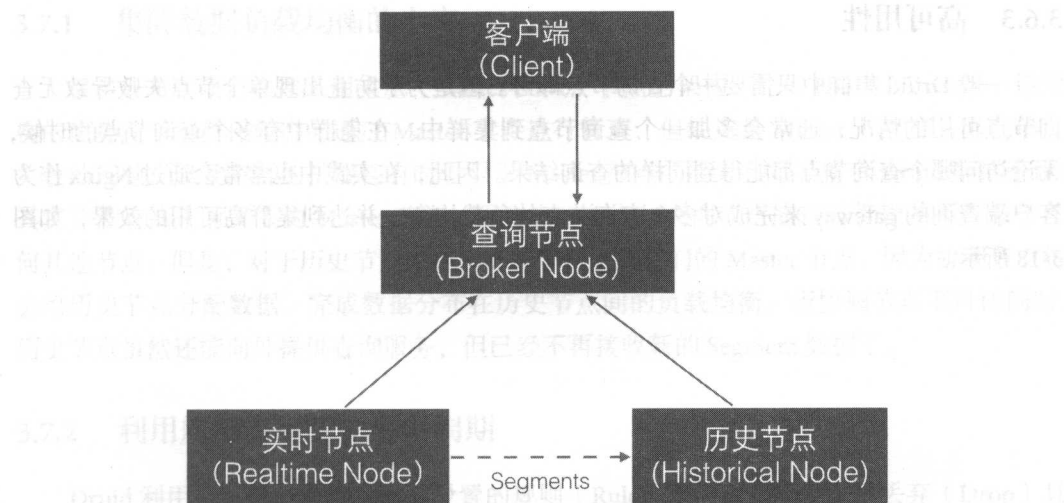


图 3-16 查询节点查询数据

3.6.2 缓存的使用

很多数据库都会利用缓存（Cache）来存储之前的查询结果，如图 3-17 所示。当相似查询再次发生时，可以直接利用之前存储在 Cache 中的数据作为部分或全部的结果，而不用再次访问库中的相关数据——这样，在 Cache 中数据的命中率较高时，查询效率也会得到明显的提高。

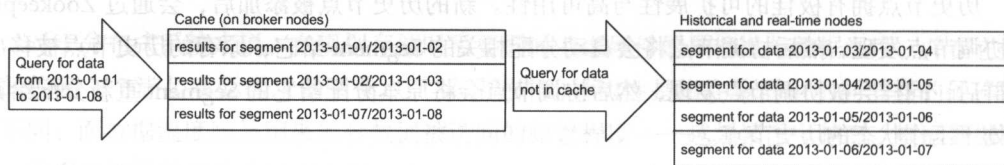


图 3-17 使用查询节点的缓存

Druid 也使用了 Cache 机制来提高自己的查询效率。Druid 提供了两类介质作为 Cache 以供选择。

- 外部 Cache，比如 Memcached。
- 本地 Cache，比如查询节点或历史节点的内存作为 Cache。

如果用查询节点的内存作为 Cache，查询的时候会首先访问其 Cache，只有当不命中的时候才会去访问历史节点与实时节点以查询数据。

3.6.3 高可用性

一般 Druid 集群中只需要一个查询节点即可。但是为了防止出现单个节点失败导致无查询节点可用的情况，通常会多加一个查询节点到集群中。在集群中有多个查询节点的时候，无论访问哪个查询节点都能得到同样的查询结果。因此，在实践中也常常会通过 Nginx 作为客户端查询的 gateway 来完成对多个查询节点的负载均衡，并达到集群高可用的效果，如图 3-18 所示。

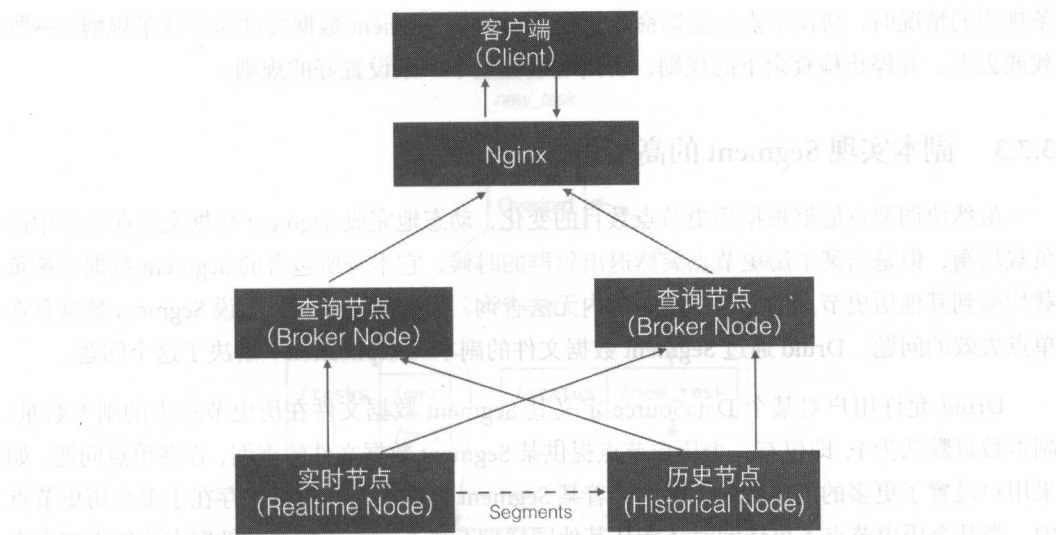


图 3-18 通过 Nginx 访问查询节点

3.7 协调节点

协调节点 (Coordinator Node) 负责历史节点的数据负载均衡, 以及通过规则管理数据的生命周期。

3.7.1 集群数据负载均衡的主宰

很多分布式项目往往采用主从 (Master-Slave) 节点的架构, 比如 HDFS、Yarn 等。该架构的优势在于集群比较容易通过 Master 节点进行管理, 但缺点是 Master 节点容易出现单点失效的问题, 以及集群的扩展性有时受限于 Master 节点的能力。对于整个 Druid 集群来说, 其实并没有真正意义上的 Master 节点, 因为实时节点与查询节点能自行管理并不听命于任何其他节点; 但是, 对于历史节点来说, 协调节点便是它们的 Master 节点, 因为协调节点将会给历史节点分配数据, 完成数据分布在历史节点间的负载均衡。当协调节点不可访问时, 历史节点虽然还能向外提供查询服务, 但已经不再接收新的 Segment 数据了。

3.7.2 利用规则管理数据生命周期

Druid 利用针对每个 DataSource 设置的规则 (Rule) 来加载 (Load) 或丢弃 (Drop) 具体的数据文件, 以管理数据生命周期。可以对一个 DataSource 按顺序添加多条规则, 对于一个 Segment 数据文件来说, 协调节点会逐条检查规则, 当碰到当前 Segment 数据文件符合某

条规则的情况时，协调节点会立即命令历史节点对该 Segment 数据文件执行这条规则——加载或丢弃，并停止检查余下的规则，否则继续检查下一条设置好的规则。

3.7.3 副本实现 Segment 的高可用性

虽然协调节点能够根据历史节点数目的变化，动态地完成 Segment 数据文件在集群中的负载均衡，但是当某个历史节点突然退出集群的时候，它本身所包含的 Segment 数据在被负载均衡到其他历史节点前的这一段时间内无法查询，因此从这个意义上说 Segment 数据存在单点失效的问题。Druid 通过 Segment 数据文件的副本（Replication）解决了这个问题。

Druid 允许用户对某个 DataSource 定义其 Segment 数据文件在历史节点中的副本数量。副本数量默认为 1，即仅有一个历史节点提供某 Segment 数据文件的查询，存在单点问题。如果用户设置了更多的副本数量，则意味着某 Segment 数据文件在集群中存在于多个历史节点中，当某个历史节点不可访问时还能从其他同样拥有该 Segment 数据文件副本的历史节点中查询到相关数据——Segment 数据文件的单点问题便迎刃而解。这个特性不仅能够应对历史节点的突发下线情况，在需要做集群升级的时候也能保证集群查询服务在此过程中不间断。

3.7.4 高可用性

对于协调节点来说，高可用性的问题十分容易解决——只需要在集群中添加若干个协调节点即可。当某个协调节点退出服务时，集群中的其他协调节点依然能够自动完成相关工作。

3.8 索引服务

除了通过实时节点生产出 Segment 数据文件外，Druid 还提供了一组名为索引服务（Indexing Service）的组件同样能够制造 Segment 数据文件。相比实时节点生产 Segment 数据文件的方式，索引服务的优点是除了对数据能够用 pull 的方式外，还支持 push 的方式；不同于手工编写数据消费配置文件的方式，可以通过 API 的编程方式来灵活定义任务配置；可以更灵活地管理与使用系统资源；可以完成 Segment 副本数量的控制；能够灵活完成跟 Segment 数据文件相关的所有操作，如合并、删除 Segment 数据文件等。

3.8.1 主从结构的架构

不同于实时节点的单节点模式，索引服务实际包含一组组件，并以主从（Master-Slave）结构作为其架构方式，其中统治节点（Overlord Node）为主节点，而中间管理者（Middle

Manager) 为从节点。索引服务架构示意图如图 3-19 所示。

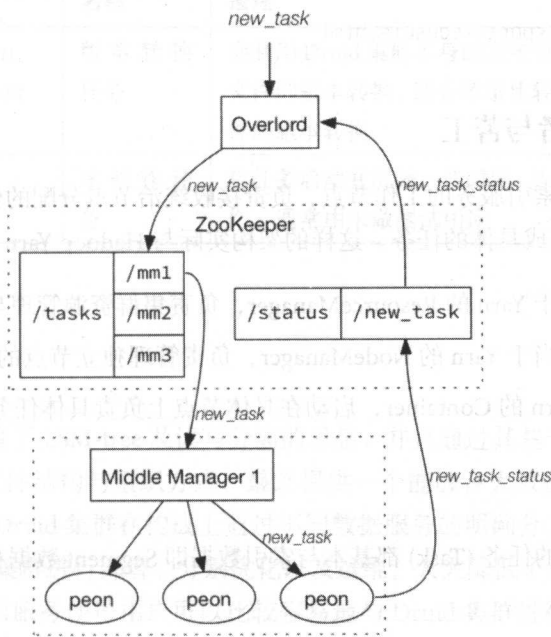


图 3-19 索引服务架构示意图

3.8.2 统治节点

统治节点作为索引服务的主节点，对外负责接收任务请求，对内负责将任务分解并下发到从节点即中间管理者上。统治节点有以下两种运行模式。

- 本地模式 (Local Mode): 默认模式。在该模式下，统治节点不仅负责集群的任务协调分配工作，也能够负责启动一些苦工 (Peon) 来完成一部分具体的任务。
- 远程模式 (Remote Mode): 在该模式下，统治节点与中间管理者分别运行在不同的节点上，它仅负责集群的任务协调分配工作，不负责完成任何具体的任务。统治节点提供 RESTful 的访问方法，因此客户端可以通过 HTTP POST 请求向统治节点提交任务。命令格式如下。

`http://<OVERLORD_IP>:<port>/druid/indexer/v1/task`

客户端也可以发出杀掉某个任务的命令。

`http://<OVERLORD_IP>:<port>/druid/indexer/v1/task/{taskId}/shutdown`

客户端可以发出查看任务状态、接收 Segment 等的任务。同时统治节点也提供了一个控制台 (Console)，因此用户可以通过浏览器轻松地了解任务与工作节点的状态。

```
http://<OVERLORD_IP>:<port>/console.html
```

3.8.3 中间管理者与苦工

中间管理者就是索引服务的工作节点，负责接收统治节点分配的任务，然后启动相关苦工即独立的 JVM 来完成具体的任务。这样的架构实际与 Hadoop Yarn 很像。

- 统治节点相当于 Yarn 的 ResourceManager，负责集群资源管理与任务分配。
- 中间管理者相当于 Yarn 的 NodeManager，负责管理独立节点的资源并接受任务。
- 苦工相当于 Yarn 的 Container，启动在具体节点上负责具体任务的执行。

3.8.4 任务

索引服务能执行的任务 (Task) 都基本与索引数据即 Segment 数据相关，下面通过表格做一个简单介绍。

父类	Type	名称	描述
创建 Segment 型	index_hadoop	Hadoop 索引任务	利用 Hadoop 集群执行 MapReduce 任务以完成 Segment 数据文件的创建，适合体量比较大的 Segment 数据文件的创建任务
创建 Segment 型	index	普通索引任务	利用 Druid 集群本身的系统资源来完成 Segment 数据文件的创建，适合体量比较小的 Segment 数据文件的创建任务
合并 Segment 型	append	附加索引任务	将若干个 Segment 数据文件首尾相连，最终合成一个 Segment 数据文件
合并 Segment 型	merge	合并索引任务	将若干个 Segment 数据文件按照指定的聚合方法合并为一个 Segment 数据文件
销毁 Segment 型	kill	销毁索引任务	彻底将指定的 Segment 数据文件从 Druid 集群包括 DeepStorage 上删除
混杂型	hadoop_convert_segment	版本转换任务	通常用来重定义 Segment 数据文件的压缩方法等要素。Hadoop 版本转换任务会利用 Hadoop 集群执行 MapReduce 任务以完成 Segment 数据文件的版本转换，适合体量比较大的 Segment 数据文件的版本转换

续表

父类	Type	名称	描述
混杂型	convert_segment	版本转换任务	会利用 Druid 集群本身的系统资源完成 Segment 数据文件的版本转换, 适合体量比较小的 Segment 数据文件的版本转换
混杂型	noop	无操作任务	将任务启动并沉睡一段时间, 并不完成具体的任何工作。通常用来做测试用途

3.9 小结

Druid 在架构上借鉴了 LSM-tree 及读写分离的思想, 并且通过其基于 DataSource 与 Segment 实现的精妙数据文件结构与组织方式, 最终提供一个能够在大数据集上做高效实时数据消费与探索的平台。Druid 集群在构成上通过不同数据服务的明确分工及协同合作, 使得用户可以比较简单地对集群进行部署、分别优化以及运维, 大大降低了集群的使用成本。同时, Druid 也通过其索引服务使得用户可以比较容易地与 Druid 集群进行交互, 完成任务的管理。

第4章

安装与配置

服务的搭建与运行是认识 Druid 的第一步，采取分布式设计的 Druid 由不同职责的节点组成，同时为了实现节点间的信息同步和服务高可用性，其还依赖一些外部组件如 Zookeeper、MySQL 和 HDFS。本章将从如下几部分重点介绍 Druid 的安装部署和配置。

- Druid 服务的安装部署（单机版用于测试学习，集群版用于生产环境）。
- 通过一个简单例子快速上手 Druid 的数据导入和查询。
- 如何搭建一个 Druid 集群以及关键配置说明。

4.1 安装准备

4.1.1 安装包简介

获取 Druid 安装包有以下几种方式。

- 源代码编译：github.com/druid-io/druid/releases，这种方式更多的是出于定制化需求的考虑，比如结合实际生产环境中的周边依赖（如依赖特定的 Hadoop 版本做持久化存储），或者是加入支持特定查询部分的优化补丁等。
- 官网安装包：druid.io/downloads.html，包含 Druid 部署运行的最基本组件。
- **Impley 组合套件（Impley Analytics Platform）**：impley.io/download，该套件包含了稳定版本的 Druid 组件、实时数据写入支持服务、图形化展示查询 Web UI 和 SQL 查询支持组件等，目的是为了更加方便、快速地部署搭建基于 Druid 的数据分析应用产品。

推荐采用这种方式来安装部署 Druid，利用组合套件可以方便、快捷地构建数据分析应用平台。

本章以安装包 `imply-1.3.1.tar.gz` 为例，解压缩后的安装包目录如下。

- **bin** —— 包含运行相关组件的脚本程序。
- **conf** —— 包含生产环境集群配置文件。
- **conf-quickstart** —— 包含单机测试版配置文件。
- **dist** —— 包含相关软件包（`druid`、`pivot`、`tranquility-server` 和 `zookeeper`）。
- **quickstart** —— 包含单机测试版快速上手相关文件。



说明：安装包 `dist` 中的 Druid 不包含实时节点。在老版本的 Druid 中，实时节点采用 `pull` 的方式实时地从消息队列如 `Kafka` 中拉取数据写入 Druid，但是由于存在单点故障等问题，故官方推荐改用索引服务（即统治节点 + 中间管理者，采用 `push` 的方式写入数据）替代实时节点在实时数据摄入方面的工作，因此关于实时节点的内容本章不再阐述。

4.1.2 安装环境

以 `Imply` 组合套件安装包为例，安装基本条件如下。

- Java 7 或者更高版本（推荐使用 Java 8）。
- NodeJS 4.x 以上版本（`PlyQL`、`Pivot` 依赖）。
- Linux 或者类 UNIX 操作系统，不支持 Windows 系统。
- 4GB 以上内存。



备注：建议生产环境部署 Druid 服务时单独申请一个特定的系统账号。该账号一方面可用于 Druid 的运行和维护；另一方面又可作为数据存储的使用账户（比如利用 `HDFS` 作为 Druid 的数据文件持久化存储层）。

以上是运行 Druid 服务的基本条件，在生产环境中搭建 Druid 集群的推荐硬件配置如下。

应用场景		CPU	内存	硬盘	SSD
小规模集群 (如 PoC 等)	所有 Druid 节点	16 核, 2.50GHz	32GB	500GB	—
大规模集群 (如生产等)	协调节点	4 核, 2.50GHz	15.0GB	20GB	—
	查询节点	16 ~ 32 核, 2.50GHz	32 ~ 244GB	20GB	推荐使用
	历史节点	16 ~ 32 核, 2.50GHz	32 ~ 244GB	200GB	推荐使用
	实时节点	16 ~ 32 核, 2.50GHz	32 ~ 244GB	100GB	推荐使用
	统治节点	4 核, 2.50GHz	15.0GB	20GB	—
	中间管理者	32 核, 2.50GHz	64 ~ 244GB	20GB	推荐使用

4.1.3 Druid 外部依赖

- **Deep Storage (数据文件存储库)**: 负责存储和加载 Druid 的数据文件 (Segment), 在生产环境中由 Deep Storage 层保障 Druid 数据文件的安全性和可用性。推荐使用 HDFS、S3, 当然 Druid 也支持其他存储组件如 Cassandra 等。
- **Metadata Storage (元数据库)**: 负责存储和管理整个系统的配置记录信息, 比如 Druid 中各个数据源、数据文件信息, 数据的加载与失效规则等。推荐使用 MySQL 或者 PostgreSQL。
- **ZooKeeper (集群状态管理服务)**: 由于 Druid 采用多节点、多角色的分布式设计, 因此管理并同步各个节点的状态信息, 以及新增节点时的服务发现功能则交给 ZooKeeper 服务来完成。

Druid 服务外部依赖模块示意图如图 4-1 所示。

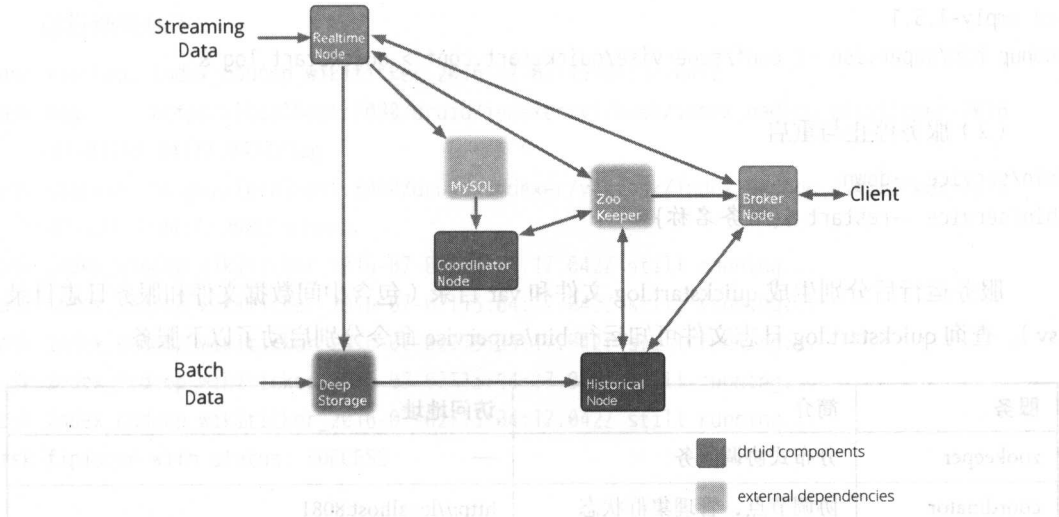


图 4-1 Druid 服务外部依赖模块示意图

4.2 简单示例

本节将以一个简单的例子作为 Druid 的入门简介，并希望通过该示例可以了解到以下几点。

- Druid 离线数据的摄入方式
- Druid 实时数据的摄入方式
- Druid 数据查询方式

4.2.1 服务运行

本节以 `imply-1.3.1.tar.gz`（Druid 版本为 0.9.1.1）为例，外部依赖采用默认配置。

- DeepStorage -> 本地存储
- Metadata Storage -> Derby
- Zookeeper -> ImPLY 安装包自带的 zk 服务

(1) 服务安装与启动

```
curl -O https://static.imply.io/release/imply-1.3.1.tar.gz
tar -xzf imply-1.3.1.tar.gz
```

```
cd imply-1.3.1
nohup bin/supervise -c conf/supervise/quickstart.conf > quickstart.log &
```

(2) 服务停止与重启

```
bin/service --down
bin/service --restart ${服务名称}
```

服务运行后分别生成 quickstart.log 文件和 var 目录（包含中间数据文件和服务日志目录 sv）。查询 quickstart.log 日志文件可知运行 bin/supervise 命令分别启动了以下服务。

服务	简介	访问地址
zookeeper	分布式协调服务	—
coordinator	协调节点，管理集群状态	http://localhost:8081
broker	查询节点，处理查询请求	http://localhost:8082/druid/v2
historical	历史节点，管理历史数据	http://localhost:8083/druid/v2
overlord	统治节点，管理数据写入任务	http://localhost:8090/console.html
middleManager	中间管理者，负责写数据处理	—
pivot	数据查询 Web UI	http://localhost:9095
tranquility-server	实时写入服务，HTTP 协议	http://localhost:8200/v1/post/datasource

4.2.2 数据导入与查询

1. 离线导入与查询

数据说明如下。

- quickstart/wikiticker-2016-06-27-sampled.json 文件包含了维基百科网站一段时间内收集到的日志数据（每条记录以一行 JSON 字符串组织）。
- quickstart/wikiticker-index.json 文件为离线写入任务的描述文件，其用 JSON 格式组织和定义了写入任务的数据源、时间信息、维度信息、指标信息等。

本示例中定义的数据源名称为 wikiticker（数据源类似于数据库中表的概念）。首先，我们尝试将这些待分析的日志数据导入到 Druid 中，代码如下。

```
bin/post-index-task --file quickstart/wikiticker-index.json
```

运行结果如下。

```
Task started: index_hadoop_wikiticker_2016-07-02T13:04:17.042Z
Task log:      http://localhost:8090/druid/indexer/v1/task/index_hadoop_wikiticker_2016-07-02T13:04:17.042Z/log
Task status:   http://localhost:8090/druid/indexer/v1/task/index_hadoop_wikiticker_2016-07-02T13:04:17.042Z/status
Task index_hadoop_wikiticker_2016-07-02T13:04:17.042Z still running...
Task index_hadoop_wikiticker_2016-07-02T13:04:17.042Z still running...
Task index_hadoop_wikiticker_2016-07-02T13:04:17.042Z still running...
Task index_hadoop_wikiticker_2016-07-02T13:04:17.042Z still running...
Task index_hadoop_wikiticker_2016-07-02T13:04:17.042Z still running...
Task finished with status: SUCCESS
```

成功写入数据后查询方式如下。

通过 Pivot 页面查询，浏览器访问地址 <http://localhost:9095>，如图 4-2 所示。

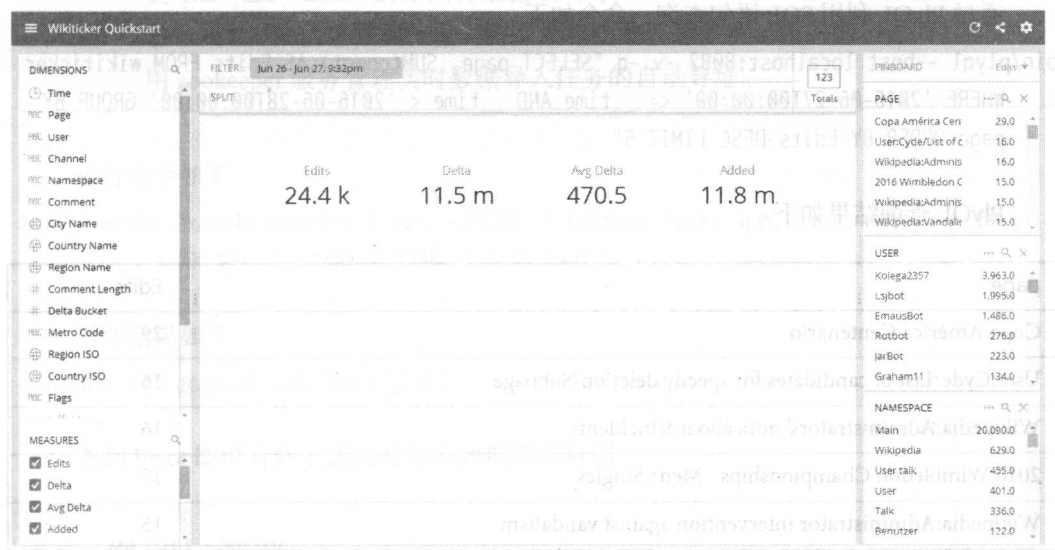


图 4-2 Pivot 查询示例

通过 HTTP POST 请求查询，quickstart/wikiticker-top-pages.json 文件包含了数据源 wiki-ticker 的 TopN 查询语句 Schema 定义信息。HTTP 查询命令如下。

```
curl -L -H'Content-Type: application/json' -XPOST --data-binary @quickstart/wikiticker-top-pages.json http://localhost:8082/druid/v2/?pretty
```

HTTP 查询结果如下。

```
[ {
  "timestamp" : "2016-06-27T00:00:11.080Z",
  "result" : [ {
    "edits" : 29,
    "page" : "Copa América Centenario"
  }, {
    "edits" : 16,
    "page" : "User:Cyde/List of candidates for speedy deletion/Subpage"
  }, {
    "edits" : 16,
    "page" : "Wikipedia:Administrators' noticeboard/Incidents"
  }
  ... ]
} ]
```

通过 PlyQL 利用 SQL 语句查询，命令如下。

```
bin/plyql --host localhost:8082 -v -q "SELECT page, SUM(count) AS Edits FROM wikiticker
WHERE '2016-06-27T00:00:00' <= __time AND __time < '2016-06-28T00:00:00' GROUP BY
page ORDER BY Edits DESC LIMIT 5"
```

PlyQL 查询结果如下。

page	Edits
Copa América Centenario	29
User:Cyde/List of candidates for speedy deletion/Subpage	16
Wikipedia:Administrators' noticeboard/Incidents	16
2016 Wimbledon Championships – Men's Singles	15
Wikipedia:Administrator intervention against vandalism	15

2. 实时导入与查询

本示例中实时数据的导入需要借助 tranquility-server 服务，如图 4-3 所示，其提供了 HTTP 协议访问接口供其他应用程序将一条或者一小批数据通过 HTTP POST 方式实时导入 Druid 中。

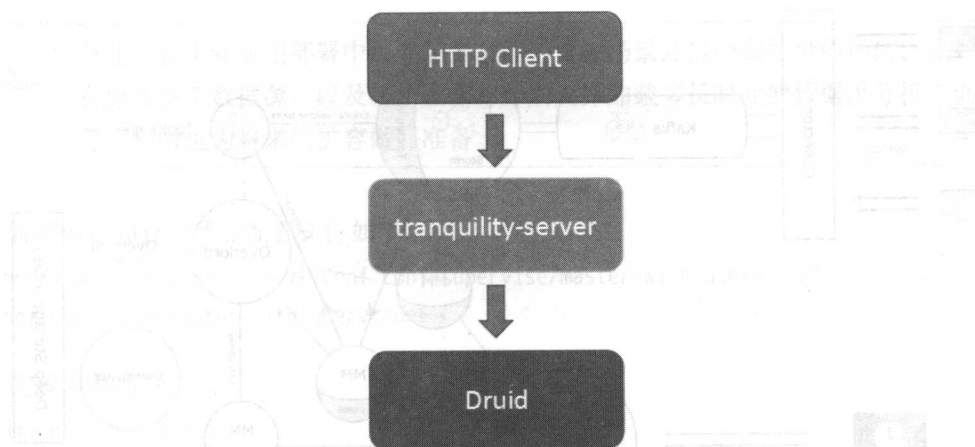


图 4-3 Druid 实时数据导入模块示意图



备注：在生产环境中通常利用流式计算引擎（如 Storm、Spark Streaming）以集成 Tranquility Client 的方式将数据实时导入 Druid 中。一方面，待分析的数据在导入 Druid 前还需要根据其特征做相应的处理清洗工作；另一方面，可以利用 zookeeper 服务管理实时数据导入任务的自动寻址。

运行命令如下。

```
bin/generate-example-metrics | curl -XPOST -H'Content-Type: application/json' --data-binary @- http://localhost:8200/v1/post/metrics
```

返回结果如下。

```
{"result":{"received":25,"sent":25}}
```

查询 Pivot 即可看到上述实时导入数据源的数据。

4.3 规划与部署

在生产环境中，Druid 可作为大数据平台的一部分与其他技术组件集成在一起工作，如图 4-4 所示。

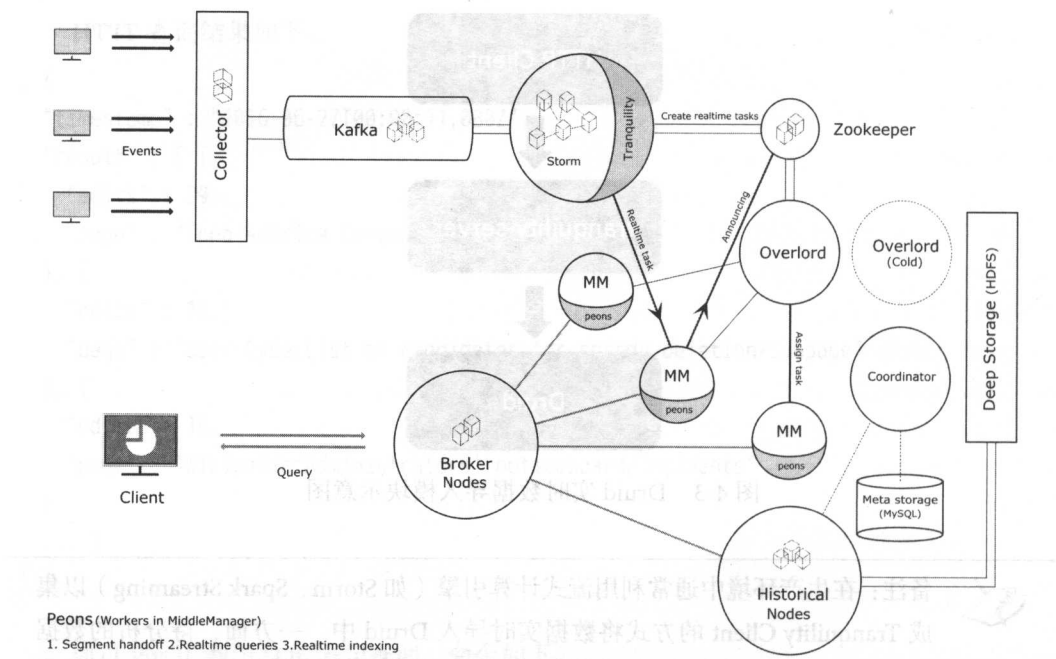


图 4-4 Druid 生产环境与其他技术组件集成示意图

由于 Druid 采用分布式设计，其中不同类型的节点各司其职，故在实际部署集群环境中首先需要对各类节点进行统一规划，从功能上划分为以下 3 个部分。

- **Master:** 管理节点，包含协调节点和统治节点，负责管理数据写入任务及容错相关处理。
- **Data:** 数据节点，包含历史节点和中间管理者，负责数据写入处理、历史数据的加载与查询。
- **Query:** 查询节点，包含查询节点和 Pivot Web 界面，负责提供数据查询接口和 Web 交互式查询功能。

在实际部署中，一方面，考虑到容错性问题，至少需要部署两个管理节点作为互备；另一方面，由于 Druid 支持横向扩展，故在使用初期考虑到机器资源有限，可以将管理节点和查询节点混合部署在同一台物理机器上，同时为了加速热点数据的查询，可以考虑加上历史节点，利用分层特性把小部分热点数据源放在管理节点所在机器上的历史节点中。而在部署机器选择上，管理节点和查询节点考虑选用多核大内存机器，比如 16 核 CPU 和 64GB 内存。由于数据节点还涉及历史数据的本地缓存，故需要更大的磁盘空间，若想获得更优的性能，推荐使用 SSD 存储设备。



备注：在实际应用部署中需要结合具体的业务场景并提前做好量的预估，比如存储多少个数据源，以及每个数据源分别支撑加载多长时间的数据供分析查询等，同时也为将来的扩容做好准备。

管理节点和查询节点配置文件如下。

```
cp conf/supervise/master-no-zk.conf conf/supervise/master-with-query.conf
vi conf/supervise/master-with-query.conf
```

配置内容如下。

```
:verify bin/verify-java
:verify bin/verify-node

broker bin/run-druid broker conf
historical bin/run-druid historical conf
pivot bin/run-pivot conf
coordinator bin/run-druid coordinator conf
!p80 overlord bin/run-druid overlord conf
```



备注：!p 加数字表示为服务关闭顺序的权重，默认值为 50，数值越大越先被关闭。

运行命令如下。

```
nohup bin/supervise -c conf/supervise/master-with-query.conf > master-with-query.log &
```

数据节点配置文件如下。

```
vi conf/supervise/data.conf
```

配置内容如下。

```
:verify bin/verify-java

historical bin/run-druid historical conf
middleManager bin/run-druid middleManager conf
```

运行命令如下。

```
nohup bin/supervise -c conf/supervise/data.conf > data.log &
```

4.4 基本配置

本节主要涉及集群的基础依赖配置和相关节点的调优选项（需要根据实际的硬件配置进行相应的优化调整）。

4.4.1 基础依赖配置

配置文件为 `conf/druid/_common/common.runtime.properties`。

（1）Zookeeper

`druid.zk.service.host=${Zookeeper集群地址}`

`druid.zk.paths.base=/druid`



说明：在 HA 的 Druid 集群配置中，所有的 Druid 节点依赖同一个 zk 集群同步节点状态信息。

（2）Metadata Storage（推荐使用 MySQL 或者 PostgreSQL）

For MySQL:

`druid.extensions.loadList=["mysql-metadata-storage"]`

`druid.metadata.storage.type=mysql`

`druid.metadata.storage.connector.connectURI=jdbc:mysql://${IP:PORT}/druid`

`druid.metadata.storage.connector.user=${USER}`

`druid.metadata.storage.connector.password=${PASSWORD}`

或者

For PostgreSQL:

`druid.extensions.loadList=["postgresql-metadata-storage"]`

`druid.metadata.storage.type=postgresql`

`druid.metadata.storage.connector.connectURI=jdbc:postgresql://${IP:PORT}/druid`

`druid.metadata.storage.connector.user=${USER}`

`druid.metadata.storage.connector.password=${PASSWORD}`

（3）Deep Storage（推荐采用 HDFS）

`druid.extensions.loadList=["druid-hdfs-storage"]`

`druid.storage.type=hdfs`

```
druid.storage.storageDirectory=hdfs://namenode.example.com:9000/druid/segments
```

```
druid.indexer.logs.type=file
```

```
druid.indexer.logs.directory=hdfs://namenode.example.com:9000/druid/indexing-logs
```



备注：采用 HDFS 作为 Deep Storage 时，离线批量导入数据任务会利用 MapReduce 加速写入处理，因此需要将生产环境中 Hadoop 对应的客户端配置文件 `core-site.xml`、`hdfs-site.xml`、`yarn-site.xml`、`mapred-site.xml` 放入 `conf/-druid/_common` 目录下。

4.4.2 数据节点配置调优

historical 与 middleManager 配置：

- JVM 内存使用 -Xmx 和 -Xms
- `druid.server.http.numThreads`
- `druid.processing.buffer.sizeBytes`
- `druid.processing.numThreads`
- `druid.query.groupBy.maxIntermediateRows`
- `druid.query.groupBy.maxResults`
- `druid.server.maxSize` 和 `druid.segmentCache.locations` (historical)
- `druid.worker.capacity` (middleManager)
- `druid.server.tier` (默认为 `_default_tier`，自定义名称可以对数据存储做分层处理)
- `druid.server.priority` (定义层对应的查询优先级)

4.4.3 查询节点配置调优

broker 相关配置：

- JVM 内存使用 -Xmx 和 -Xms
- `druid.server.http.numThreads`
- `druid.cache.sizeInBytes`
- `druid.processing.buffer.sizeBytes`

- `druid.processing.numThreads`
- `druid.query.groupBy.maxIntermediateRows`
- `druid.query.groupBy.maxResults`

配置说明如下。

- Druid 中查询节点和历史节点都提供了针对查询的本地 LRU 缓存机制，在配置方面 `broker` 和 `historical` 只需要在一种节点上开启缓存即可，推荐小集群（<20 台机器）在查询节点上开启查询缓存，大集群在历史节点上开启查询缓存。
- 并发性能的调优更多的是通过调整相关处理的线程数来实现，查询涉及多个数据文件（Segment）的计算时，通常一个数据文件对应一个处理线程。

详细配置说明：<http://druid.io/docs/latest/configuration/index.html>

4.5 集群节点配置示例

4.5.1 节点规划

这里以实际应用部署为例，机器类型分为以下两种。

- Master 机器：64GB 内存、16 核 CPU、250GB 磁盘空间。
- Data 机器：64GB 内存、24 核 CPU、1TB 磁盘空间。

初始搭建 Druid 集群选取 Master 机器 2 台、Data 机器 3 台，其中 Master 机器作为管理和查询节点，Data 机器作为数据节点，分别部署服务如下。



备注：考虑到时区问题，所有 Druid 相关的服务时区需要设置一致，这里统一配置为 UTC+0800。

1. 全局 Common 配置（`common.runtime.properties`）

```
# Extensions
```

```
druid.extensions.directory=dist/druid/extensions
```

```
druid.extensions.hadoopDependenciesDir=dist/druid/hadoop-dependencies
```

```
druid.extensions.loadList=["druid-lookups-cached-global", "druid-histogram", "druid-datasketches", "mysql-metadata-storage", "druid-hdfs-storage"]
```

```
# Log all runtime properties on startup
druid.startup.logging.logProperties=true

# Zookeeper
druid.zk.service.host=${ZK_IPs} //Zookeeper 集群IP地址，不同的IP地址间用逗号(,)连接
druid.zk.paths.base=/druid

# Metadata storage
druid.metadata.storage.type=mysql
druid.metadata.storage.connector.connectURI=jdbc:mysql://${MYSQL_IP:PORT}/druid?
    characterEncoding=UTF-8
druid.metadata.storage.connector.user=${MYSQL_USER}
druid.metadata.storage.connector.password=${MYSQL_PASSWORD}

# Deep storage
druid.storage.type=hdfs
druid.storage.storageDirectory=hdfs://${SEGMENTS_PATH}

druid.indexer.logs.type=hdfs
druid.indexer.logs.directory=hdfs://${INDEXING_LOGS_PATH}

# Service discovery
druid.selectors.indexing.serviceName=druid/overlord
druid.selectors.coordinator.serviceName=druid/coordinator

# Monitoring
druid.monitoring.monitors=["com.metamx.metrics.JvmMonitor"]
druid.emitter=logging
druid.emitter.logging.logLevel=info
```



备注：后续的章节会详细介绍 Druid 服务的相关指标信息和监控方式。

2. 全局 log4j2.xml 日志配置

```
<?xml version="1.0" encoding="UTF-8" ?>
<Configuration status="WARN">
```



```

<Appenders>
  <Console name="Console" target="SYSTEM_OUT">
    <PatternLayout pattern="%d{ISO8601} %p [%t] %c - %m%n"/>
  </Console>
</Appenders>
<Loggers>
  <AsyncLogger name="io.druid.curator.inventory.CuratorInventoryManager" level="
    info" additivity="false">
    <AppenderRef ref="Console"/>
  </AsyncLogger>
  <AsyncLogger name="io.druid.client.BatchServerInventoryView" level="info"
    additivity="false">
    <AppenderRef ref="Console"/>
  </AsyncLogger>
  <!-- Make extra sure nobody adds logs in a bad way that can hurt performance -->
  <AsyncLogger name="io.druid.client.ServerInventoryView" level="info" additivity
    ="false">
    <AppenderRef ref="Console"/>
  </AsyncLogger>
  <AsyncLogger name="com.metamx.http.client.pool.ChannelResourceFactory" level="
    info" additivity="false">
    <AppenderRef ref="Console"/>
  </AsyncLogger>
  <Root level="info">
    <AppenderRef ref="Console"/>
  </Root>
</Loggers>
</Configuration>

```



备注：对于频繁打印日志的类，可以参考如上所示将其日志打印方式设置为异步方式。

4.5.2 Master 机器配置

Master 机器 2 台，作为管理节点可相互用作 HA 支撑，同时又承担了部分数据查询的工作。

1. 协调节点

jvm.config:

```
-server  
-Xms3g  
-Xmx3g  
-Duser.timezone=UTC+0800  
-Dfile.encoding=UTF-8  
-Djava.io.tmpdir=var/tmp  
-Djava.util.logging.manager=org.apache.logging.log4j.jul.LogManager
```

runtime.properties:

```
druid.service=druid/coordinator  
druid.host=${IP_ADDR} //部署机器IP地址  
druid.port=8081  
druid.coordinator.startDelay=PT30S  
druid.coordinator.period=PT30S
```

2. 统治节点

jvm.config:

```
-server  
-Xms4g  
-Xmx4g  
-XX:NewSize=256m  
-XX:MaxNewSize=256m  
-XX:+UseConcMarkSweepGC  
-XX:+PrintGCDetails  
-XX:+PrintGCTimeStamps  
-Duser.timezone=UTC+0800  
-Dfile.encoding=UTF-8  
-Djava.io.tmpdir=var/tmp  
-Djava.util.logging.manager=org.apache.logging.log4j.jul.LogManager
```

runtime.properties:

```
druid.service=druid/overlord  
druid.host=${IP_ADDR} //部署机器IP地址  
druid.port=8090
```

```
druid.indexer.queue.startDelay=PT30S
druid.indexer.runner.type=remote
druid.indexer.storage.type=metadata
```



备注：统治节点分配写入数据任务给 `middleManager` 的负载模式需要服务启动后通过动态 HTTP POST 请求的方式修改，默认方式为 `fillCapacity`，推荐改为 `equalDistribution` 方式。

```
curl -L -H'Content-Type: application/json' -X POST -d '{"selectStrategy":
{"type": "equalDistribution"}}' http://${OVERLORD_IP:PORT}/druid/
indexer/v1/worker
```

查询当前统治节点任务分配方式设置如下。

```
curl -L -H'Content-Type: application/json' -X GET http://${OVERLORD_IP:PORT}/druid/
indexer/v1/worker/history?count=10
```

3. 查询节点

`jvm.config:`

```
-server
-Xms24g
-Xmx24g
-XX:NewSize=6g
-XX:MaxNewSize=6g
-XX:MaxDirectMemorySize=32g
-XX:+UseConcMarkSweepGC
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
-Duser.timezone=UTC+0800
-Dfile.encoding=UTF-8
-Djava.io.tmpdir=var/tmp
-Djava.util.logging.manager=org.apache.logging.log4j.jul.LogManager
```

`runtime.properties:`

```
druid.service=druid/broker
druid.host=${IP_ADDR} //部署机器IP地址
druid.port=8082
```

```
# HTTP server threads
druid.broker.http.numConnections=20 //查询节点与历史节点通信连接池大小
druid.broker.http.readTimeout=PT5M
druid.server.http.numThreads=50 //HTTP请求处理线程数

# Processing threads and buffers
druid.processing.buffer.sizeBytes=2147483647 //每个处理线程的中间结果缓存大小，单位
    为Byte
druid.processing.numThreads=15 //处理线程数

# Query cache
druid.broker.cache.useCache=true
druid.broker.cache.populateCache=true
druid.broker.cache.unCacheable=[]
druid.cache.sizeInBytes=6000000000 //JVM堆内LRU缓存大小，单位为Byte

# Query config
druid.broker.balancer.type=connectionCount //查询节点请求历史节点方式，有random和
    connectionCount两种连接方式
```

4. 历史节点（加载热点数据）

jvm.config:

```
-server
-Xms12g
-Xmx12g
-XX:NewSize=4g
-XX:MaxNewSize=4g
-XX:MaxDirectMemorySize=16g
-XX:+UseConcMarkSweepGC
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
-Duser.timezone=UTC+0800
-Dfile.encoding=UTF-8
-Djava.io.tmpdir=var/tmp
-Djava.util.logging.manager=org.apache.logging.log4j.jul.LogManager
```

runtime.properties:

```
druid.service=druid/historical
druid.host=${IP_ADDR} //部署机器IP地址
druid.port=8083

# HTTP server threads
druid.server.http.numThreads=50 //HTTP请求处理线程数

# Processing threads and buffers
druid.processing.buffer.sizeBytes=1073741824 //每个查询处理线程的中间结果缓存大小，
    单位为Byte
druid.processing.numThreads=15 //查询处理线程数

# Segment storage
druid.segmentCache.locations=[{"path":"var/druid/segment-cache","maxSize"
    "\:100000000000"}] //Segment本地加载路径与最大存储空间大小，单位为Byte
druid.server.maxSize=100000000000 //最大存储空间大小，该值只用作Coordinator调配
    Segment加载的依据

# Query cache
druid.historical.cache.useCache=false
druid.historical.cache.populateCache=false

# Tier
druid.server.tier=hot //自定义数据层名称，默认为_default_tier，不同数据层的Segment数
    据无法相互复制
druid.server.priority=10 //自定义数据层优先级，默认值为0，值越大优先级越高，该功能
    用于冷热数据层的划分
```

4.5.3 Data 机器配置

Data 机器 3 台，作为数据节点负责数据处理，Shared nothing 架构。

1. 历史节点

jvm.config:

```
-server
-Xms12g
```

```
-Xmx12g
-XX:NewSize=6g
-XX:MaxNewSize=6g
-XX:MaxDirectMemorySize=32g
-XX:+UseConcMarkSweepGC
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
-Duser.timezone=UTC+0800
-Dfile.encoding=UTF-8
-Djava.io.tmpdir=var/tmp
-Djava.util.logging.manager=org.apache.logging.log4j.jul.LogManager
```

runtime.properties:

```
druid.service=druid/historical
druid.host=${IP_ADDR} //部署机器IP地址
druid.port=8083
```

HTTP server threads

```
druid.server.http.numThreads=50 //HTTP请求处理线程数
```

Processing threads and buffers

```
druid.processing.buffer.sizeBytes=1073741824 //每个处理线程的中间结果缓存大小，单位
为Byte
```

```
druid.processing.numThreads=23 //处理线程数
```

Segment storage

```
druid.segmentCache.locations=[{"path":"var/druid/segment-cache","maxSize"
"\:500000000000"}] //Segment本地加载路径与最大存储空间大小，单位为Byte
```

```
druid.server.maxSize=500000000000 //最大存储空间大小，该值只用作Coordinator调配
Segment加载的依据
```

Query cache

```
druid.historical.cache.useCache=false
```

```
druid.historical.cache.populateCache=false
```

2. 中间管理者

jvm.config:

```
-server
-Xms64m
-Xmx64m
-XX:+UseConcMarkSweepGC
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
-Duser.timezone=UTC+0800
-Dfile.encoding=UTF-8
-Djava.io.tmpdir=var/tmp
-Djava.util.logging.manager=org.apache.logging.log4j.jul.LogManager
```

runtime.properties:

```
druid.service=druid/middlemanager
druid.host=${IP_ADDR} //部署机器IP地址
druid.port=8091
```

```
# Number of tasks per middleManager
```

```
druid.worker.capacity=23 //最大可接受处理的任务数，默认为处理器数-1
```

```
# Task launch parameters(peons)
```

```
druid.indexer.runner.javaOpts=-server -Xmx3g -XX:MaxDirectMemorySize=24g -XX:+UseG1GC -
XX:MaxGCPauseMillis=100 -Duser.timezone=UTC+0800 -Dfile.encoding=UTF-8 -Djava.util.
logging.manager=org.apache.logging.log4j.jul.LogManager
```

```
druid.indexer.task.baseTaskDir=var/druid/task
```

```
druid.indexer.task.restoreTasksOnRestart=true
```

```
# HTTP server threads
```

```
druid.server.http.numThreads=50 //HTTP请求处理线程数
```

```
# Processing threads and buffers
```

```
druid.processing.buffer.sizeBytes=1073741824 //每个处理线程的中间结果缓存大小，单位
为Byte
```

```
druid.processing.numThreads=23 //处理线程数
```

```
# Hadoop indexing
```



```
druid.indexer.task.hadoopWorkingPath=var/druid/hadoop-tmp
druid.indexer.task.defaultHadoopCoordinates=["org.apache.hadoop:hadoop-client:2.2.0"] //
```

Hadoop版本需要根据具体情况来配置

4.6 小结

安装与配置是探索 Druid 的第一步。本章介绍了如何一步步搭建和运行 Druid 服务，并在最后给出了一种在生产环境中搭建 Druid 集群的参考方案。结合实践来说，其实 Druid 在配置方面还有很多值得结合实际场景探索挖掘的地方，故没有最优的配置，只有最适合的配置。下一步就可以基于搭建好的服务，即刻动手去尝试 Druid 在数据分析方面的能力，希望读者能在实践中找到适合具体业务的解决方案。

第5章

数据摄入

本章将介绍 Druid 数据摄入的基本原理，并通过实例讲解如何通过不同的方式将数据摄入到 Druid 数据库中。同时，也会展示如何通过使用 Lambda 架构构建一个能解决 Druid 时间窗口限制的数据摄入系统。最后，本章将介绍其他一些关于 Druid 数据摄入的重要知识点。

5.1 数据摄入的两种方式

数据摄入是指为了立即使用数据或者将数据持久化，从而获取和导入数据的过程。数据摄入包括流式和批量两种方式：流式数据摄入主要指数据源一边产生数据一边导入；而批量数据摄入则主要指不同的数据块（chunk）周期性导入。由于存在大量的不同格式的数据，因此，如何以合理的速度以及过程摄入数据对于业务系统来说具有非常大的挑战。本章将详细描述如何有效地把外部数据摄入到 Druid 的各种方式和技巧。对于高级技巧，例如如何保证数据一致性等，将在第 6 章中进行详细阐述。

Druid 的数据摄入也分为流式和批量两种方式，而针对不同类型的数据，Druid 将外部数据源分为两种形式。

5.1.1 流式数据源

流式数据源，指的是持续不断地生产数据的数据源。非常典型的例子有消息队列、日志文件等。

对于流式数据的摄取，Druid 提供了两种方式，分别是 Push（推送）和 Pull（拉取）。采用 Pull 方式摄取数据，需要启动一个实时节点（Realtime Node），通过不同的 Firehose 摄入不同的流式数据。Firehose 可以认为是 Druid 接入不同数据源的 Adapter。例如从 Kafka 中摄取数据，就可以使用 KafkaFirehose；从 RabbitMQ 中摄取数据，就可以使用 RabbitMQFirehose。采用 Push 方式摄取数据，需要使用 Druid 索引服务（Indexing Service）。索引服务会启动一个 HTTP 服务，数据通过调用这个 HTTP 服务推送到 Druid 系统。

5.1.2 静态数据源

静态数据源，指的是数据已经生产完毕，不会有新数据产生的数据源。静态数据通常表现为离线数据，比如文件系统中的文件。静态数据可以通过实时节点摄入，也可以通过索引服务启动一个任务进行摄取。

整个数据摄取流程的数据流入示意图如图 5-1 所示。

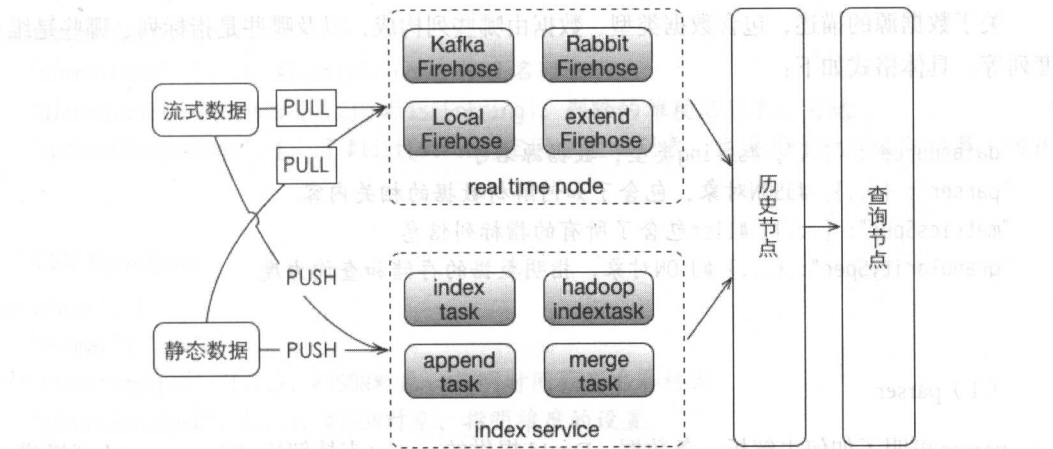


图 5-1 整个数据摄取流程的数据流入示意图

5.2 流式数据摄取

正如前文所述，流式数据可以通过两种方式摄取：通过实时节点以 Pull 方式摄取；通过索引服务以 Push 方式摄取。

5.2.1 以 Pull 方式摄取

以 Pull 方式摄取，需要启动一个实时节点。在启动实时节点的过程中，需要一个配置文件去指定数据摄取的相关参数，在 Druid 系统中这个配置文件称为 Ingestion Spec。

Ingestion Spec 是一个 JSON 格式的文本，由三个部分组成。

```
{
  "dataSchema" : {...}, #JSON对象，指明数据源格式、数据解析、维度等信息
  "ioConfig" : {...}, #JSON对象，指明数据如何在 Druid 中存储
  "tuningConfig" : {...} #JSON对象，指明存储优化配置
}
```

以下是三个部分的简述。

1. DataSchema

关于数据源的描述，包含数据类型、数据由哪些列构成，以及哪些是指标列、哪些是维度列等。具体格式如下：

```
{
  "datasource": "...", #string类型，数据源名字
  "parser": {...}, #JSON对象，包含了如何解析数据的相关内容
  "metricsSpec": [...], #list包含了所有的指标列信息
  "granularitySpec": {...} #JSON对象，指明数据的存储和查询力度
}
```

(1) parser

parser 声明了如何去解析一条数据。Druid 提供的 parser 支持解析 string、protobuf 格式；同时社区贡献了一些插件以支持其他数据格式，比如 avro 等。本书主要涉及 string 格式。parser 的数据结构如下：

```
"parser": {
  "type": "...", #string数据类型
  "parseSpec": {...} #JSON对象
}
```

parseSpec 指明了数据源格式，比如维度列表、指标列表、时间戳列名等。接下来简述在日常开发中运用得比较多的三种数据格式（JSON, CSV, TSV）。

JSON ParseSpec:

```
"parseSpec": {
  "format": "json",
  "timestampSpec": {...}, #JSON对象, 指明时间戳列名和格式
  "dimensionsSpec": {...}, #JSON对象, 指明维度的设置
  "flattenSpec": {...} #JSON对象, 若JSON有嵌套层级, 则需要指定
}
```

其中 timestampSpec 如下:

```
"timestampSpec": {
  "column": "...", #string, 时间戳列名
  "format": "...", #iso|millis|posix|auto|Joda, 时间戳格式, 默认值为 auto
}
```

dimensionsSpec 如下:

```
"dimensionsSpec": {
  "dimensions": [...], #list[string], 维度名列表
  "dimensionExclusions": [...], #list[string], 剔除的维度名列表; 可选
  "spatialDimensions": [...] #list[string]空间维度名列表, 主要用于地理几何运算; 可选
}
```

CSV ParseSpec:

```
"parseSpec": {
  "format": "csv",
  "timestampSpec": {...}, #JSON对象, 指明时间戳列名和格式
  "dimensionsSpec": {...}, #JSON对象, 指明维度的设置
  "columns": [...], #list[string], CSV数据列名
  "listDelimiter": "...", #string, 多值维度列, 数据分隔符; 可选
}
```

其中 timestampSpec 和 dimensionsSpec 请参考前文。

TSV ParseSpec:

```
"parseSpec": {
  "format": "tsv",
  "timestampSpec": {...}, #JSON 对象, 指明时间戳列名和格式
  "dimensionsSpec": {...}, #JSON 对象, 指明维度的设置
  "columns": [...], #list[string], TSV数据列名
}
```

```

"listDelimiter": "...", #string, 多值维度列, 数据分隔符; 可选
"delimiter": "...", #string, 数据分隔符, 默认值为\t; 可选
}

```

(2) metricsSpec

metricsSpec 是一个 JSON 数组, 指明所有的指标列和所使用的聚合函数。数据格式如下:

```

"metricsSpec": [
{
    "type": "...", #count|longSum等聚合函数类型
    "fieldName": "...", #string, 聚合函数运用的列名; 可选
    "name": "...", #string, 聚合后指标列名
},
...
]

```

下面介绍几种常用的聚合函数, 一些复杂的聚合函数会在第 6 章中介绍。

- **count Aggregator:** 统计满足查询过滤条件的数据行数。注意, 这个行数和初始的摄入数据行数是不同的, 因为 Druid 会对原始数据进行聚合, 这个行数就是指聚合后的行数。
- **longSum Aggregator:** 对所有满足查询过滤条件的行做求和运算。应用于数据类型是整数的列。
- **longMin Aggregator:** 对所有满足查询过滤条件的行求最小值。应用于数据类型是整数的列。
- **longMax Aggregator:** 对所有满足查询过滤条件的行求最大值。应用于数据类型是整数的列。
- **doubleSum Aggregator:** 对所有满足查询过滤条件的行做求和运算。应用于数据类型是浮点数的列。
- **doubleMin Aggregator:** 对所有满足查询过滤条件的行求最小值。应用于数据类型是浮点数的列。
- **doubleMax Aggregator:** 对所有满足查询过滤条件的行求最大值。应用于数据类型是浮点数的列。
- **GranularitySpec:** 指定 Segment 的存储粒度和查询粒度。具体的数据格式如下:

```
"granularitySpec": {
  "type": "uniform",
  "segmentGranularity": "...", #string, Segment 的存储粒度 HOUR、DAY等
  "queryGranularity": "...", #string, 最小查询粒度 MINUTE、HOUR等
  "intervals": [ ... ] #摄取的数据的时间段，可以有多个值；
                        #可选，对于流式数据Pull方式可以忽略
}
```

2. ioConfig

ioConfig 指明了真正的具体的数据源，它的格式如下：

```
"ioConfig": {
  "type": "realtime",
  "firehose": {...}, #指明数据源，例如本地文件、Kafka等
  "plumber": "realtime"
}
```

不同的 firehose 的格式不太一致，下面以 Kafka 为例，说明 firehose 的格式。

```
{
  "firehose": {
    "consumerProps": {
      "auto.commit.enable": "false",
      "auto.offset.reset": "largest",
      "fetch.message.max.bytes": "1048586",
      "group.id": "druid-example",
      "zookeeper.connect": "localhost:2181",
      "zookeeper.connection.timeout.ms": "15000",
      "zookeeper.session.timeout.ms": "15000",
      "zookeeper.sync.time.ms": "5000"
    },
    "feed": "wikipedia",
    "type": "kafka-0.8"
  }
}
```


3. tuningConfig

这部分配置可以用于优化数据摄取的过程，具体格式如下：

```
"tuningConfig": {
  "type": "realtime",
  "maxRowsInMemory": "...", #在存盘之前内存中最大的存储行数，指的是聚合后的行数
  "windowPeriod": "...",    #最大可容忍时间窗口，超过窗口，数据丢弃
  "intermediatePersistPeriod": "...", #多长时间数据临时存盘一次
  "basePersistDirectory": "...",    #临时存盘目录
  "versioningPolicy": "...",        #如何为 Segment 设置版本号
  "rejectionPolicy": "...",        #数据丢弃策略
  "maxPendingPersists": ...,      #最大同时存盘请求数，达到上限，摄取将会暂停
  "shardSpec": {...},            #分片设置
  "buildV9DDirectly": ...,        #是否直接构建 v9版本的索引
  "persistThreadPriority": "...",  #存盘线程优先级
  "mergeThreadPriority": "...",   #存盘归并线程优先级
  "reportParseExceptions": ...,   #是否汇报数据解析错误
}
```

以上是实时数据以 Pull 方式摄取的相关配置。下面我们以一个网站对其用户行为数据进行摄取的任务为例，说明如何使用 Pull 方式摄取用户的行为数据。

5.2.2 用户行为数据摄取案例

在一个网站的运营工作中，网站的运营人员常常需要对用户行为进行分析，而进行分析前的重要工作之一便是对用户行为数据进行格式定义与摄取方式的确定。当使用 Druid 来完成用户行为数据摄取的工作时，我们可以使用一个 DataSource 来存储用户行为，而使用类似如下的 JSON 数据来定义这个 DataSource 的格式。

```
{
  "age": "90+",
  "category": "3C",
  "city": "Beijing",
  "count": 1,
  "event_name": "browse_commodity",
  "timestamp": "2016-07-04T13:30:21.563Z",
  "user_id": 123
}
```

接下来,对于用户行为数据的摄入部分,我们以 Druid 实时节点从 Kafka 中 Pull 数据为例来进行说明。

首先,若以 Pull 的方式摄取数据,则需要启动一个实时节点。而启动实时节点,则需要一个 Spec 配置文件。Spec 配置文件是一个 JSON 文件。我们要把上述的 JSON 格式的行为数据通过实时节点导入到 Druid 系统,该 Spec 配置文件的内容如下:

```
[
  {
    "dataSchema": {
      "dataSource": "dianshang_order",
      "granularitySpec": {
        "queryGranularity": "MINUTE",
        "segmentGranularity": "HOUR",
        "type": "uniform"
      },
      "metricsSpec": [
        {
          "fieldName": "count",
          "name": "count",
          "type": "longSum"
        }
      ],
      "parser": {
        "parseSpec": {
          "dimensionsSpec": {
            "dimensionExclusions": [],
            "dimensions": [
              "event_name",
              "user_id",
              "age",
              "city",
              "commodity",
              "category"
            ],
            "spatialDimensions": []
          },
          "format": "json",
          "timestampSpec": {
```

```

        "column": "timestamp",
        "format": "auto"
    },
    "type": "string"
},
{
    "ioConfig": {
        "firehose": {
            "consumerProps": {
                "auto.commit.enable": "false",
                "auto.offset.reset": "largest",
                "fetch.message.max.bytes": "1048586",
                "group.id": "druid-example",
                "zookeeper.connect": "localhost:2181",
                "zookeeper.connection.timeout.ms": "15000",
                "zookeeper.session.timeout.ms": "15000",
                "zookeeper.sync.time.ms": "5000"
            },
            "feed": "dianshang_order",
            "type": "kafka-0.8"
        },
        "plumber": {
            "type": "realtime"
        },
        "type": "realtime"
    },
    "tuningConfig": {
        "basePersistDirectory": "/tmp/realtime/basePersist",
        "intermediatePersistPeriod": "PT10m",
        "maxRowsInMemory": 75000,
        "rejectionPolicy": {
            "type": "serverTime"
        },
        "type": "realtime",
        "windowPeriod": "PT10m"
    }
}
]

```

在使用上述 Spec 配置文件启动实时节点后，实时节点就会自动地从 Kafka 通过 Pull 的方式摄取数据。

我们会在后面的章节中介绍如何查询摄入到 Druid 系统的数据。

5.2.3 以 Push 方式摄取

以 Push 方式摄取，需要索引服务，所以要先启动中间管理者（Middle Manager）和统治节点（Overlord Node）（具体请参见第4章）。

1. 启动索引任务

启动索引任务需要向索引服务中的统治节点发送一个 HTTP 请求，并向该请求 POST 一份 Ingestion Spec。我们接着用用户行为数据摄取的例子，Ingestion Spec 如下：

```
{
  "spec": {
    "dataSchema": {
      "dataSource": "dianshang_order",
      "granularitySpec": {
        "queryGranularity": "MINUTE",
        "segmentGranularity": "HOUR",
        "type": "uniform"
      },
      "metricsSpec": [
        {
          "fieldName": "count",
          "name": "count",
          "type": "longSum"
        }
      ],
      "parser": {
        "parseSpec": {
          "dimensionsSpec": {
            "dimensionExclusions": [],
            "dimensions": [
              "event_name",
              "user_id",
              "age",
              "city",
              "commodity",
```

```

        "category": "category",
        "spatialDimensions": [],
    },
    "format": "json",
    "timestampSpec": {
        "column": "timestamp",
        "format": "auto"
    },
    "type": "map"
},
},
"ioConfig": {
    "firehose": {
        "serviceName": "dianshang_order",
        "type": "receiver"
    },
    "type": "realtime"
},
},
"tuningConfig": {
    "basePersistDirectory": "/rc/data/druid/realtime/basePersist",
    "intermediatePersistPeriod": "PT10m",
    "maxRowsInMemory": 500000,
    "rejectionPolicy": {
        "type": "serverTime"
    },
    "type": "realtime",
    "windowPeriod": "PT10m"
},
"indexer": {
    "type": "index_realtime"
}
}

```

下面以这份 Ingestion Spec 启动索引任务。

```
curl -X 'POST' -H 'Content-Type:application/json' -d @my-index-task.json OVERLORD_IP:
PORT/druid/indexer/v1/task
```

这样会把任务分配给中间管理者，用于接收数据。

2. 发送数据

```
curl -X 'POST' -H 'Content-Type:application/json' -d '[{"timestamp":"2016-07-13T06:17:29","event_name":"浏览商品","user_id":1,"age":"90+","city":"Beijing","commodity":"xxxxx","category":"3c","count":1}]' peonhost:port/druid/worker/v1/chat/dianshang_order/push-events
```

5.2.4 索引服务任务相关管理接口

索引服务启动的相关任务可以通过相应的接口进行管理。

1. 提交任务

```
curl -X 'POST' -H 'Content-Type:application/json' -d @my-index-task.json OVERLORD_IP:PORT/druid/indexer/v1/task
```

2. 查看任务状态

```
curl http://<OVERLORD_IP>:<port>/druid/indexer/v1/task/{taskId}/status
```

得到查询任务的状态如下：

```
...
{
  "task": "index_realtime_dianshang_order_0_2016-07-13T05:44:17.189Z_ndnklphj",
  "status": {
    "id": "index_realtime_dianshang_order_0_2016-07-13T05:44:17.189Z_ndnklphj",
    "status": "RUNNING",
    "duration": -1
  }
},
{
  "task": "index_realtime_dianshang_order_0_2016-07-13T05:44:17.189Z_ndnklphj",
  "status": {
    "id": "index_realtime_dianshang_order_0_2016-07-13T05:44:17.189Z_ndnklphj",
    "status": "RUNNING",
    "duration": -1
  }
}
...
```

3. 查看 Segment 信息

```
curl http://10.24.199.8:8090/druid/indexer/v1/task/index_realtime_dianshang_order_0_2016-07-13T05:44:17.189Z_ndnklphj/segments
```

返回的信息如下:

```
[
  {
    "binaryVersion": 9,
    "dataSource": "dianshang_order",
    "dimensions": "event_name,user_id,age,city,commodity,category",
    "identifier": "dianshang_order_2016-07-13T06:00:00.000Z_2016-07-13T07:00:00.000Z_2016-07-13T06:20:41.042Z",
    "interval": "2016-07-13T06:00:00.000Z/2016-07-13T07:00:00.000Z",
    "loadSpec": {
      "path": "/druid/segments/dianshang_order/20160713T060000.000Z_20160713T070000.000Z/2016-07-13T06_20_41.042Z/0/index.zip",
      "type": "hdfs"
    },
    "metrics": "count",
    "shardSpec": {
      "type": "none"
    },
    "size": 4382,
    "version": "2016-07-13T06:20:41.042Z"
  },
  {
    "binaryVersion": 9,
    "dataSource": "dianshang_order",
    "dimensions": "event_name,user_id,age,city,commodity,category",
    "identifier": "dianshang_order_2016-07-13T05:00:00.000Z_2016-07-13T06:00:00.000Z_2016-07-13T05:45:02.324Z",
    "interval": "2016-07-13T05:00:00.000Z/2016-07-13T06:00:00.000Z",
    "loadSpec": {
      "path": "/druid/segments/dianshang_order/20160713T050000.000Z_20160713T060000.000Z/2016-07-13T05_45_02.324Z/0/index.zip",
      "type": "hdfs"
    },
  },
```



```
    "metrics": "count",
    "shardSpec": {
      "type": "none"
    },
    "size": 4394,
    "version": "2016-07-13T05:45:02.324Z"
  }
}
```

4. 关闭任务

curl -X 'POST' http://<OVERLORD_IP>:<port>/druid/indexer/v1/task/{taskId}/shutdown

5. Overlord 控制台

Druid 提供了一个控制台，可以查看任务的状态、日志、Worker 等，如图 5-2 所示。

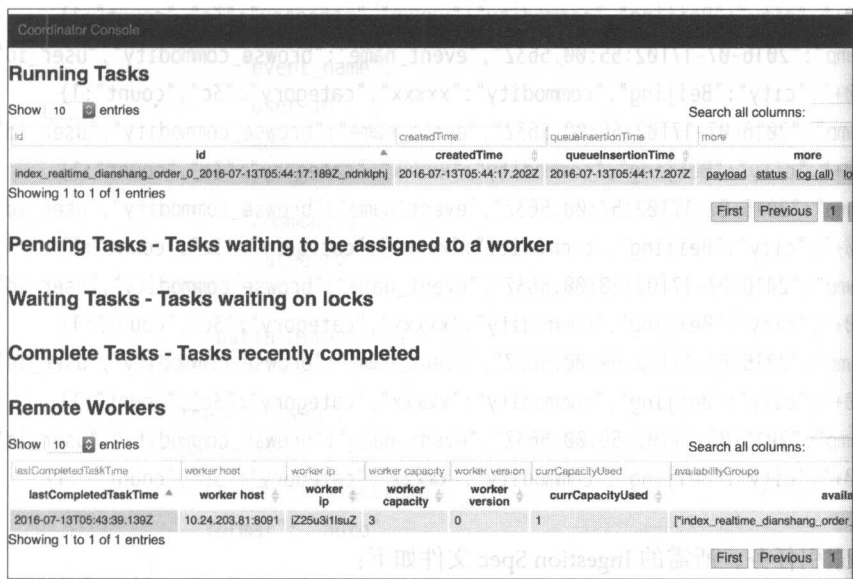


图 5-2 控制台

5.3 静态数据批量摄取

5.3.1 以索引服务方式摄取

同样，也可以通过索引服务方式批量摄取数据。我们仍然需要通过统治节点提交一个索引任务。例如，把用户行为数据批量导入到系统中。用户行为数据示例如下：

```
{
  "timestamp": "2016-07-17T02:50:00.563Z", "event_name": "browse_commodity", "user_id": 1, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 2
}
{
  "timestamp": "2016-07-17T02:51:00.563Z", "event_name": "browse_commodity", "user_id": 1, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1
}
{
  "timestamp": "2016-07-17T02:52:00.563Z", "event_name": "browse_commodity", "user_id": 1, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1
}
{
  "timestamp": "2016-07-17T02:53:00.563Z", "event_name": "browse_commodity", "user_id": 1, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1
}
{
  "timestamp": "2016-07-17T02:54:00.563Z", "event_name": "browse_commodity", "user_id": 1, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1
}
{
  "timestamp": "2016-07-17T02:55:00.563Z", "event_name": "browse_commodity", "user_id": 1, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1
}
{
  "timestamp": "2016-07-17T02:56:00.563Z", "event_name": "browse_commodity", "user_id": 1, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1
}
{
  "timestamp": "2016-07-17T02:57:00.563Z", "event_name": "browse_commodity", "user_id": 1, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1
}
{
  "timestamp": "2016-07-17T02:58:00.563Z", "event_name": "browse_commodity", "user_id": 1, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1
}
{
  "timestamp": "2016-07-17T02:59:00.563Z", "event_name": "browse_commodity", "user_id": 1, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1
}
{
  "timestamp": "2016-07-17T02:59:00.563Z", "event_name": "browse_commodity", "user_id": 1, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1
}
```

启动索引任务，所需的 Ingestion Spec 文件如下：

```
{
  "spec": {
    "dataSchema": {
      "dataSource": "dianshang_order",
      "granularitySpec": {
        "intervals": [
          "2016-07-17/2016-07-18"
```

```

    ],
    "queryGranularity": "MINUTE",
    "segmentGranularity": "HOUR",
    "type": "uniform"
  },
  "metricsSpec": [
    {
      "fieldName": "count",
      "name": "count",
      "type": "longSum"
    }
  ],
  "parser": {
    "parseSpec": {
      "dimensionsSpec": {
        "dimensionExclusions": [],
        "dimensions": [
          "event_name",
          "user_id",
          "age",
          "city",
          "commodity",
          "category"
        ],
        "spatialDimensions": []
      },
      "format": "json",
      "timestampSpec": {
        "column": "timestamp",
        "format": "auto"
      }
    },
    "type": "string"
  }
},
"ioConfig": {
  "firehose": {
    "baseDir": "/rc/data/druid/tmp/test-data",

```

```

        "filter": "*.json",
        "type": "local"
    },
    "type": "index"
}
},
"tuningConfig": {
    "basePersistDirectory": "/rc/data/druid/realtime/basePersist",
    "intermediatePersistPeriod": "PT10m",
    "maxRowsInMemory": 500000,
    "rejectionPolicy": {
        "type": "none"
    },
    "type": "index",
    "windowPeriod": "PT10m"
},
"type": "index"
}

```

5.3.2 以 Hadoop 方式摄取

Druid Hadoop Index Job 支持从 HDFS 上读取数据，并摄入 Druid 系统中。启动一个 Hadoop Index Job，需要 POST 一个请求到 Druid 统治节点。沿用之前的案例，启动一个 Hadoop Index Job，向统治节点 POST 的启动任务的数据如下：

```

{
    "spec": {
        "dataSchema": {
            "dataSource": "dianshang_order",
            "granularitySpec": {
                "intervals": [
                    "2016-07-19/2016-07-20"
                ],
                "queryGranularity": "MINUTE",
                "segmentGranularity": "HOUR",
                "type": "uniform"
            },
            "metricsSpec": [

```

```

        {
            "fieldName": "count",
            "name": "count",
            "type": "longSum"
        }
    ],
    "parser": {
        "parseSpec": {
            "columns": [
                "timestamp",
                "event_name",
                "user_id",
                "age",
                "city",
                "commodity",
                "category",
                "count"
            ],
            "dimensionsSpec": {
                "dimensionExclusions": [],
                "dimensions": [
                    "event_name",
                    "user_id",
                    "age",
                    "city",
                    "commodity",
                    "category"
                ],
                "spatialDimensions": []
            },
            "format": "csv",
            "timestampSpec": {
                "column": "timestamp",
                "format": "auto"
            }
        },
        "type": "hadoopyString"
    }
}

```

```
    },
    "ioConfig": {
      "inputSpec": {
        "paths": "/tmp/dianshang_order.json",
        "type": "static"
      },
      "type": "hadoop"
    },
    "tuningConfig": {
      "type": "hadoop"
    }
  },
  "type": "index_hadoop"
}
```

启动任务：

```
curl -X 'POST' -H 'Content-Type:application/json' -d @hadoop-index-task.json http://10.24.199.8:8090/druid/indexer/v1/task
```

测试数据如下：

2016-07-19T08:36:29,浏览商品,1,90+,Beijing,xxxxx,3c,1
2016-07-19T12:36:29,浏览商品,1,90+,Beijing,yyyyy,3c,1
2016-07-19T16:36:29,浏览商品,1,90+,Beijing,zzzzz,3c,1
2016-07-19T23:36:29,浏览商品,1,90+,Beijing,aaaaa,3c,1

Druid 会提交一个 MapReduce 任务到 Hadoop 系统，所以这种方式非常适合大批量的数据摄入，如图 5-3 所示。

Cluster Metrics															
Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	V-Cores Used	V-Cores Total	V-Cores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
45	0	2	43	7	4 GB	175.78 GB	0 B	7	24	0	3	0	0	0	0
Scheduler Metrics															
Scheduler Type				Scheduling Resource Type				Minimum Allocation				Maximum Allocation			
Capacity Scheduler				[MEMORY]				<memory256, vCores:1>				<memory8192, vCores:8>			
Show 20 <input type="text" value="entries"/>															
ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI	Blacklisted Nodes				
application_1467728642584_0045	rc	dianshang_order-index-generator-Optional.ch[2016-07-19T00:00:00.000Z/2016-07-20T00:00:00.000Z]	MAPREDUCE	default	Sat Aug 20 16:45:56 +0800 2016	Sat Aug 20 16:46:35 +0800 2016	FINISHED	SUCCEEDED	<div></div>	History	N/A				
application_1467728642584_0044	rc	dianshang_order-index-generator-Optional.ch[2016-07-19T00:00:00.000Z/2016-07-20T00:00:00.000Z]	MAPREDUCE	default	Sat Aug 20 16:30:03 +0800 2016	Sat Aug 20 16:30:45 +0800 2016	FINISHED	SUCCEEDED	<div></div>	History	N/A				
application_1467728642584_0043	rc	dianshang_order-index-generator-Optional.ch[2016-07-19T00:00:00.000Z/2016-07-20T00:00:00.000Z]	MAPREDUCE	default	Sat Aug 20 16:08:45 +0800 2016	Sat Aug 20 16:09:32 +0800 2016	FINISHED	SUCCEEDED	<div></div>	History	N/A				

图 5-3 Hadoop MapReduce 批量摄入

从 Druid Coordinator Console 页面我们可以看到相关的 Segment 已经生成,如图 5-4 所示。



图 5-4 生成 Segment

5.4 流式与批量数据摄取的结合

我们都知道, Druid 在摄取时需要设置一个时间窗口,在时间窗口之外的数据,将会丢弃。我们如何将这部分丢弃的数据重新摄取进 Druid 系统中,以提高数据的准确性?通常的做法是把数据保存起来,等待重新摄取。目前,比较流行的处理方法是 Lambda 架构。

5.4.1 Lambda 架构

Lambda 是实时处理框架 Storm 的作者 Nathan Marz 提出的用于同时处理离线和实时数据的架构理念。Lambda 架构(LA)旨在满足一个稳定的大规模数据处理系统所需的容错性、低延迟、可扩展的特性。LA 的可行性和必要性基于如下假设和原则。

- 任何数据系统可定义为: $query = functional(all\ data)$ 。
- 人为容错性(Human Fault-Tolerance): 数据是易丢失的。
- 数据不可变性(Data Immutability): 数据是只读的,不再变化。
- 重新计算(Recomputation): 因为上面两个原则,运行函数重新计算结果是可能的。

LA 基本框架如图 5-5 所示。

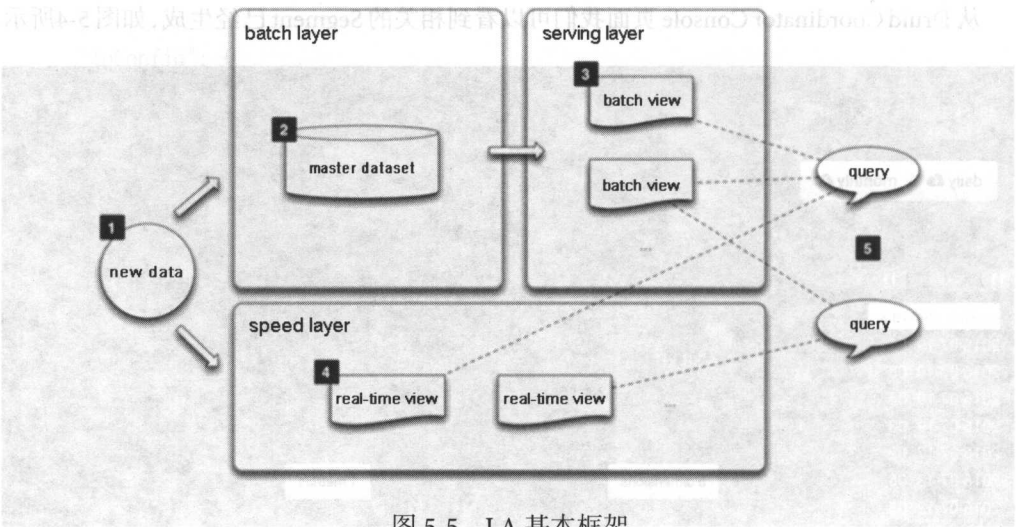


图 5-5 LA 基本框架

该架构具有如下特点。

- 所有新数据分别分发到批处理层和实时处理层。
- 批处理层有两个功能：管理主要的数据（该类数据的特点是只能增加，不能更新）；为下一步计算出批处理视图做预计算。
- 服务层计算出批处理视图中的数据做索引，以提供低延时，即席查询。
- 实时处理层仅处理实时数据，并为服务层提供查询服务。
- 任何查询都可以通过实时处理层和批处理层的查询结果合并得到。

从以上论述我们可以知道，Druid 本身就是一个典型的 Lambda 架构系统。Druid 有实时节点和历史节点，任何查询都是聚合实时节点和历史节点的数据得到查询结果。那么，我们如何在 Druid 系统之外采用 Lambda 架构的思维去解决时间窗口的问题呢？

5.4.2 解决时间窗口问题

Druid 在摄取数据时，对于超出时间窗口的数据会直接丢弃，这对于某些要求数据准确性的系统来说，是不可接受的。那么就需要重新摄入这部分数据，参考 Lambda 的思想，实现方式如图 5-6 所示。

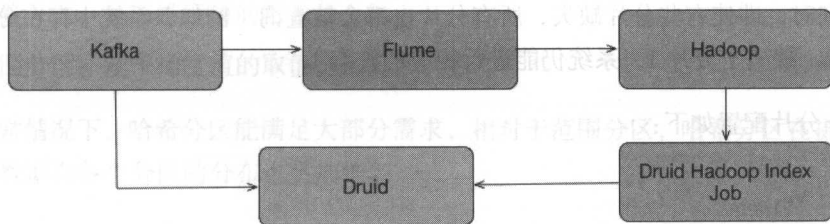


图 5-6 参考 Lambda 思想的 Druid 应用架构

流程如下。

- (1) 源数据都进入 Kafka。
- (2) 数据通过实时节点或者索引服务进入 Druid 中。
- (3) Kafka 的数据通过 Flume 备份到 Hadoop。
- (4) 定时或者发现有数据丢失时，通过 Druid Hadoop Index Job 重新摄入数据。

5.5 数据摄取的其他重要知识

5.5.1 数据分片

Druid 数据是以时间分片的，然而，当数据集中出现在某个时间段的时候，就会出现某些 Segment 过大的情况，进而加大加载时间，影响查询效率。Druid 通过数据分片与复制，使数据分布到更多的 Druid 节点，以提高并行查询的效率。我们先来看数据分片部分。

Druid 数据分片都是通过 Ingestion Spec 的 tuningConfig 设置的。对于实时、批量摄取，设置的方式有些许区别，下面分别进行论述。

1. 实时节点数据分片

流式拉取数据的方式需要启动实时节点，启动实时节点可以通过 tuningConfig 部分的 shardSpec 指定分片方式。目前支持两种分片方式，即 Linear 和 Numbered。

(1) Linear 分片

Linear 分片具有如下优势。

- 添加新的实时节点时，不用更改原实时节点的配置。

- 查询时，即使有些分片缺失，所有分片也都会被查询（例如，系统中存在分片 0、分片 2，缺失了分片 1，系统仍能进行相应的查询）。

Linear 分片配置如下：

```
"shardSpec": {  
  "type": "linear",  
  "partitionNum": 0  
}
```

（2）Numbered 分片

Numbered 分片方式，要求必须所有的分片都存在，才能提供查询。当然，很多时候这是必要的。与 Linear 分片不同的是，Numbered 分片还必须指定分片总数。

Numbered 分片配置如下：

```
"shardSpec": {  
  "type": "numbered",  
  "partitionNum": 0,  
  "partitions": 2  
}
```

2. Druid Index Job 数据分片

批量摄取方式，都是通过启动 Druid Index Job 来实现的。启动 Index Job，可以通过 `tuningConfig` 设置该任务摄取数据的分片方式。设置方式如下：

```
"tuningConfig": {  
  "type": "index",  
  "targetPartitionSize": 5000000,  
  "numShards": -1  
}
```

`targetPartitionSize` 和 `numShards` 是两种不同的分片方式，只能设置一个（不能都不等于 -1），其中 `targetPartitionSize` 是通过设置分片大小，计算出分片个数；`numShards` 则直接设定分片个数。

3. Druid Hadoop Index Job 数据分片

对于 Hadoop Index Job 数据分片，同样是通过 `tuningConfig` 部分的 `partitionsSpec` 设置的。目前，支持如下两种分区方式。

- 哈希分区：基于维度值的哈希值分区。
- 范围分区：基于维度值的取值范围分区。

在通常情况下，哈希分区能满足大部分需求，相对于范围分区，哈希分区在摄取速度上会更快，数据在各个分区的分布也更加均匀。

(1) 哈希分区

哈希分区配置如下：

```
"partitionsSpec": {  
  "type": "hashed",  
  "targetPartitionSize": 5000000  
}
```

更多的配置项如下：

配置项	描述	是否必需
type	分片类型	"hashed"
targetPartitionSize	分区目标行数，尽量使分区大小为 500MB~1GB	targetPartitionSize 和 numShards 只能设置一个
numShards	分区个数	targetPartitionSize 和 numShards 只能设置一个
partitionDimensions	基于分区的维度，仅仅配合 numShards 使用，设置了 targetPartitionSize，该项被忽略	否

(2) 范围分区

范围分区配置如下：

```
"partitionsSpec": {  
  "type": "dimension",  
  "targetPartitionSize": 5000000  
}
```

更多的配置项如下：

配置项	描述	是否必需
type	分片类型	"dimension"
targetPartitionSize	分区目标行数，尽量使分区大小为 500MB~1GB	是
maxPartitionSize	分区最大行数，默认最大值为 1.5 倍 targetPartitionSize	否
partitionDimension	基于分区的维度，没有设置，会自动选择一个维度	否
assumeGrouped	是否假设数据已经预先按时间和维度分组	否

4. 数据分区样例

我们以 Hadoop Index Job 分区为例，仍使用用户行为数据摄取案例。

以如下的 Spec 启动任务。

```
{
  "spec": {
    "dataSchema": {
      "dataSource": "dianshang_order",
      "granularitySpec": {
        "intervals": [
          "2016-08-19/2016-08-20"
        ],
        "queryGranularity": "MINUTE",
        "segmentGranularity": "HOUR",
        "type": "uniform"
      },
    },
    "metricsSpec": [
      {
        "fieldName": "count",
        "name": "count",
        "type": "longSum"
      }
    ],
    "parser": {
      "parseSpec": {
        "columns": [
```

```

        "timestamp",
        "event_name",
        "user_id",
        "age",
        "city",
        "commodity",
        "category",
        "count"
    ],
    "dimensionsSpec": {
        "dimensionExclusions": [],
        "dimensions": [
            "event_name",
            "user_id",
            "age",
            "city",
            "commodity",
            "category"
        ],
        "spatialDimensions": []
    },
    "format": "csv",
    "timestampSpec": {
        "column": "timestamp",
        "format": "auto"
    }
},
"type": "hadoopyString"
}

},
"ioConfig": {
    "inputSpec": {
        "paths": "/tmp/dianshang_order.json",
        "type": "static"
    },
    "type": "hadoop"
},
"tuningConfig": {

```

```
"partitionsSpec": {
  "numShards": 2,
  "type": "hashed"
},
"type": "hadoop"
}
},
"type": "index_hadoop"
}
```

我们可以从 Coordinator Console 看到分区信息，如图 5-7 所示。

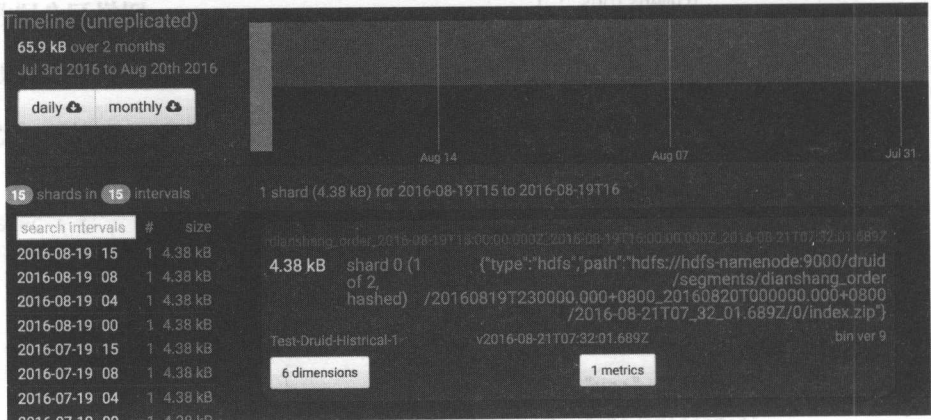


图 5-7 分区信息

从图 5-7 可以看到，这是两个分区中的第一个分区，存储在 Test-Druid-Histrcal-1 这台机器上。

5.5.2 数据复制

Druid 数据复制可以分为两个方面。

- DeepStorage：一般使用 HDFS、S3 等，这些系统本身就有副本复制能力，保证数据不会丢失。
- Druid 系统内部数据复制：Druid 存储的基本单位是 Segment，我们可以通过 Coordinator 设置 Segment 在 Druid 系统内部的副本因子，如图 5-8 所示。

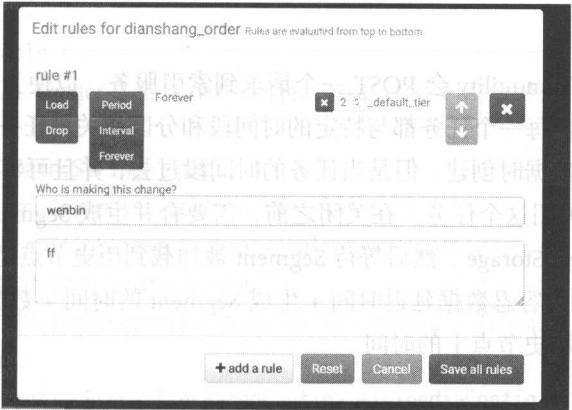


图 5-8 通过规则更改副本因子

设置后，Coordinator 会调度历史节点将副本加载到系统中，如图 5-9 所示。

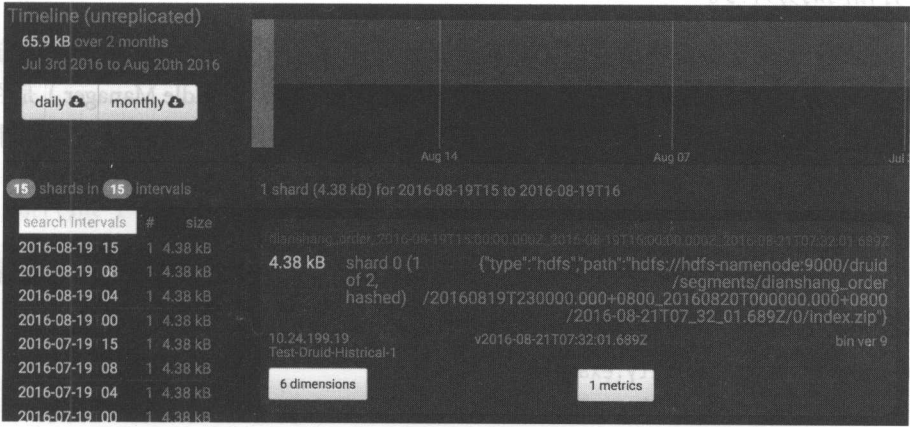


图 5-9 加载副本

从图 5-9 可以看到，之前只有 Test-Druid-Histrcial-1 这台机器存储的 Segment，现在多了一台 10.24.199.19 机器存储这个 Segment。

5.5.3 索引服务之 Tranquility

在上面内容中，我们讲述了如何通过索引服务摄取数据，然而索引服务的 API 太过底层，运用起来比较麻烦。Tranquility 对索引服务的 API 进行了封装，可以方便地创建任务、处理分片、复制、服务发现以及无缝的数据结构调整。

1. 任务创建

在通常情况下，Tranquility 会 POST 一个请求到索引服务，以便为每一个 Segment 创建一个实时摄取的任务。每一个任务都与特定的时间段和分区相关。任务会在 Tranquility 收到某一时间段的第一条数据时创建。但是当任务的时间段过去，并且可容忍的数据延迟窗口过去时，Druid 就开始关闭这个任务。在关闭之前，需要合并生成 Segment 并移交，移交是指把 Segment 存储到 DeepStorage，然后等待 Segment 被加载到历史节点上。所以，任务生存时间 = 任务时间窗口 + 可容忍数据延迟时间 + 生成 Segment 的时间 + 数据存储到 DeepStorage 的时间 + 数据加载到历史节点上的时间。

2. 分片与复制

Tranquility 通过对一个特定的时间段创建多个任务来实现分片，这些任务都会有一个不同的分片号（partitionNum）。同样，Tranquility 也是通过多个任务实现复制的，不同的是，这些任务有相同的分片号。

Tranquility 会周期性地将任务提交到索引服务。在通常情况下，由于时间窗口的存在，新任务在旧任务结束之前就已经提交，所以要考虑中间管理者（Middle Manager）是否有足够的资源运行这么多任务。中间管理者至少能支持的任务数量是： $2 \times \text{分片数} \times \text{复制副本数}$ 。

3. API

以下是 Tranquility 官方提供的样例代码，可以看到，其实读取了 example.json 配置文件，然后根据 example.json 的配置启动任务，最后通过 sender 异步发送数据。

```
package com.metamx.tranquility.example;
import com.google.common.collect.ImmutableMap;
import com.metamx.common.logger.Logger;
import com.metamx.tranquility.config.DataSourceConfig;
import com.metamx.tranquility.config.PropertiesBasedConfig;
import com.metamx.tranquility.config.TranquilityConfig;
import com.metamx.tranquility.druid.DruidBeams;
import com.metamx.tranquility.tranquilizer.MessageDroppedException;
import com.metamx.tranquility.tranquilizer.Tranquilizer;
import com.twitter.util.FutureEventListener;
import org.joda.time.DateTime;
import scala.runtime.BoxedUnit;
import java.io.InputStream;
import java.util.Map;
```



```

        if (e instanceof MessageDroppedException) {
            log.warn(e, "Dropped message: %s", obj);
        } else {
            log.error(e, "Failed to send message: %s", obj);
        }
    }
}

);
}
} finally {
    sender.flush();
    sender.stop();
}
}
}

```

当然，如果需要更灵活地控制启动任务的配置，可以通过更底层的 API 来实现，如下所示：

```

List<String> dimensions = DataSourceConfiguration.getDimensions(command);

// API
AggregatorFactory[] aggregatorFactories = DataSourceConfiguration.getAggregators(command);
List<AggregatorFactory> aggregators = Arrays.asList(aggregatorFactories);
DruidConfig config = new DruidConfig();

druidService = DruidBeams
    .builder(timestamper)
    .curator(curator)
    .discoveryPath(discoveryPath)
    .location(
        DruidLocation.create(
            indexService,
            firehosePattern,
            dataSource
        )
    )
    .timestampSpec(timestampSpec)

```

```

.rollup(DruidRollup.create(DruidDimensions.specific(dimensions), aggregators,
    QueryGranularity.MINUTE))
.tuning(ClusteredBeamTuning
    .builder()
    .segmentGranularity(Granularity.HOUR)
    .windowPeriod(new Period("PT10M"))
    .partitions(Integer.parseInt(config.getProperty(dataSource + ".partitions")))
    .replicants(Integer.parseInt(config.getProperty(dataSource + ".replicants")))
    .build())
).buildJavaService();

List<Map<String, Object>> events;
druidService.apply(events);

```

可以看到，通过 API 可以指定维度、指标以及分片与复制的配置等，这样可以支持动态调整配置变化等高阶需求。

5.5.4 高基数维度优化

Druid 高效的一个重要原因是聚合，而对于基数十分大的维度，会严重降低聚合的效率，所以需要对基数大的维度进行调优。尽量不用、少用基数大的维度，或者通过离散化的方式减少基数，而在某些业务场景下，我们可以使用一些优化算法。

1. Cardinality aggregator

对于计算一个维度基数这样的需求，我们可以使用 Cardinality aggregator 来满足。Cardinality aggregator 基于 HyperLogLog 算法，但是 Cardinality aggregator 只是在查询阶段进行了优化，所以并不能减少存储的容量，查询效率也会比在摄取阶段就进行优化的 HyperUnique aggregator 低。Cardinality aggregator 的用法如下：

```

{
  "type": "cardinality",
  "name": "<output_name>",
  "fieldNames": [ <dimension1>, <dimension2>, ... ],
  "byRow": <false | true> # (optional, defaults to false)
}

```

(1) Cardinality by value

在默认情况下，基数计算基于维度的值，下面举例说明。

- 单维度——按值求基，相当于如下 SQL:

```
SELECT COUNT(DISTINCT(dimension)) FROM <datasource>
```

- 多维度——按值求基，相当于如下 SQL:

```
SELECT COUNT(DISTINCT(value)) FROM (
  SELECT dim_1 as value FROM <datasource>
  UNION
  SELECT dim_2 as value FROM <datasource>
  UNION
  SELECT dim_3 as value FROM <datasource>
)
```

(2) Cardinality by row

按行求基，相当于如下 SQL:

```
SELECT COUNT(*) FROM ( SELECT DIM1, DIM2, DIM3 FROM <datasource> GROUP BY DIM1, DIM2,
  DIM3 )
```

2. HyperUnique aggregator

正如前文所述，Cardinality aggregator 并没有在摄取阶段进行优化。如果业务不需要对高基数维度进行过滤、分组的操作，则完全可以在摄取阶段就进行优化，以获得更少的存储容量和更快的查询速度。仍使用之前的用户行为数据摄取案例，user id 就是一个基数很大的维度，而通常我们需要统计的是 UV 这样的指标，那么完全可以通过 HyperUnique aggregator 满足。

在摄取阶段，可以进行如下配置：

```
{
  "spec": {
    "dataSchema": {
      "dataSource": "dianshang_order",
      "granularitySpec": {
        "intervals": [
          "2016-08-28/2016-08-29"
```

```

    ],
    "queryGranularity": "MINUTE",
    "segmentGranularity": "HOUR",
    "type": "uniform"
  },
  "metricsSpec": [
    {
      "fieldName": "count",
      "name": "count",
      "type": "longSum"
    },
    {
      "fieldName": "user_id",
      "name": "unique_user_id",
      "type": "hyperUnique"
    }
  ],
  "parser": {
    "parseSpec": {
      "dimensionsSpec": {
        "dimensionExclusions": [],
        "dimensions": [
          "event_name",
          "age",
          "city",
          "commodity",
          "category"
        ],
        "spatialDimensions": [],
        "format": "json",
        "timestampSpec": {
          "column": "timestamp",
          "format": "auto"
        }
      },
      "type": "string"
    }
  }
}

```



```

    },
    "ioConfig": {
      "firehose": {
        "baseDir": "/rc/data/druid/tmp/test-data",
        "filter": "*.json",
        "type": "local"
      },
      "type": "index"
    }
  },
  "tuningConfig": {
    "basePersistDirectory": "/rc/data/druid/realtime/basePersist",
    "intermediatePersistPeriod": "PT10m",
    "maxRowsInMemory": 500000,
    "rejectionPolicy": {
      "type": "none"
    },
    "type": "index",
    "windowPeriod": "PT10m"
  },
  "type": "index"
}

```

样例数据如下:

```

{"timestamp": "2016-08-28T08:50:00.563Z", "event_name": "browse_commodity", "user_id": 1, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 2}
{"timestamp": "2016-08-28T08:50:00.563Z", "event_name": "browse_commodity", "user_id": 2, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1}
{"timestamp": "2016-08-28T08:50:00.563Z", "event_name": "browse_commodity", "user_id": 1, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1}
{"timestamp": "2016-08-28T08:50:00.563Z", "event_name": "browse_commodity", "user_id": 3, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1}
{"timestamp": "2016-08-28T08:50:00.563Z", "event_name": "browse_commodity", "user_id": 2, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1}
{"timestamp": "2016-08-28T08:50:00.563Z", "event_name": "browse_commodity", "user_id": 4, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1}
{"timestamp": "2016-08-28T08:50:00.563Z", "event_name": "browse_commodity", "user_id": 5, "age": "90+", "city": "Beijing", "commodity": "xxxxx", "category": "3c", "count": 1}

```

```
{
  "timestamp": "2016-08-28T08:50:00.563Z",
  "event_name": "browse_commodity",
  "user_id": 6,
  "age": "90+",
  "city": "Beijing",
  "commodity": "xxxxx",
  "category": "3c",
  "count": 1
}
{
  "timestamp": "2016-08-28T08:50:00.563Z",
  "event_name": "browse_commodity",
  "user_id": 9,
  "age": "90+",
  "city": "Beijing",
  "commodity": "xxxxx",
  "category": "3c",
  "count": 1
}
{
  "timestamp": "2016-08-28T08:50:00.563Z",
  "event_name": "browse_commodity",
  "user_id": 8,
  "age": "90+",
  "city": "Beijing",
  "commodity": "xxxxx",
  "category": "3c",
  "count": 1
}
{
  "timestamp": "2016-08-28T08:50:00.563Z",
  "event_name": "browse_commodity",
  "user_id": 7,
  "age": "90+",
  "city": "Beijing",
  "commodity": "xxxxx",
  "category": "3c",
  "count": 1
}
```

我们可以通入如下查询，查询 UV:

```
{
  "aggregations": [
    {
      "fieldName": "unique_user_id",
      "name": "unique_user_id",
      "type": "hyperUnique"
    }
  ],
  "dataSource": "dianshang_order",
  "granularity": "day",
  "intervals": [
    "2016-08-27/2016-08-29"
  ],
  "queryType": "timeseries"
}
```

得到结果如下:

```
[
  {
    "result": {"unique_user_id": 9.019833517963864},
    "timestamp": "2016-08-28T00:00:00.000Z"
  }
]
```

5.6 小结

通过本章的介绍,相信读者已经对 Druid 的数据摄入有了比较全面的认识,也已经了解了各种数据摄入方法的适用场景,同时清楚了提高摄入性能的有效方式。无论对于何种数据平台,数据摄入几乎都是首先要考虑的工作,也是一个可以被持续优化的重要环节,而 Druid 的摄入手段也一样会不断地发展和完善,因此我们可以对此保持持续关注。

第6章

数据查询

前面的章节分别介绍了数据摄入、Data Schema 定义以及从 Hadoop 中摄入数据，本章将介绍 Druid 的数据查询过程以及查询语法。Druid 提供了 HTTP REST 风格的查询接口。用户对数据的查询通过 HTTP 请求发送到查询节点（Broker Node），然后查询节点转发至历史节点（Historical Node）或实时节点（Realtime Node）处理。我们可以使用 curl 命令进行测试，一个典型的 curl 命令如下：

```
curl -X POST '⟨broker_host⟩:⟨port⟩/druid/v2/?pretty' -H \
'Content-Type:application/json' -d @⟨query_json_file⟩
```

- queryable_host:port 为查询节点的 IP 地址和端口，往往系统中会配置多个查询节点，每个查询节点提供的服务都是相同的，可以任选一个进行学习测试。
- query_json_file 为 POST 到查询节点的查询请求。

Druid 包含多种查询类型，如对用户摄入 Druid 的数据进行 TopN, Timeseries, GroupBy, Select, Search 等方式的查询，也可以查询一个数据源的 timeBoundary, segmentMetadata, dataSourceMetadata 等。

6.1 查询过程

查询节点接收外部 Client 的查询请求，并根据查询中指定的 interval 找出相关的 Segment，然后找出包含这些 Segment 的实时节点和历史节点，再将请求分发给相应的实时节点和历史节点，最后将来自实时节点和历史节点的查询结果合并后返回给调用方。其中，查询节点通

过 Zookeeper 来发现历史节点和实时节点的存活状态。图 6-1 展示了在系统架构中查询请求数据如何流动，以及哪些节点涉入其中。

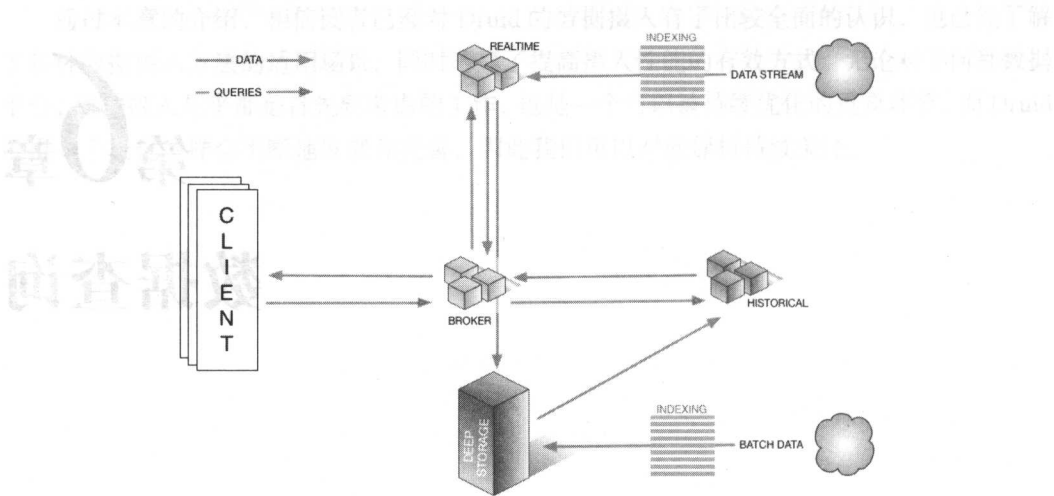


图 6-1 Druid 查询流程图

查询过程如下：

- （1）查询请求首先进入查询节点，查询节点将与已知存在的 Segment 进行匹配查询。
- （2）查询节点选择一组可以提供所需要的 Segment 的历史节点和实时节点，将查询请求分发到这些机器上。
- （3）历史节点和实时节点都会进行查询处理，然后返回结果。
- （4）查询节点将历史节点和实时节点返回的结果合并，返回给查询请求方。

6.2 组件

在介绍具体的查询之前，先介绍各种查询都会用到的基本组件（Component），如 Filter、Aggregator、Post-Aggregator、Query、Interval 和 Context 等。每一种基本组件都包含更多的详细类型。

6.2.1 Filter

Filter，即过滤器，在查询语句中是一个 JSON 对象，用来对维度进行筛选，表示维度满足 Filter 的行是我们需要的数据，类似于 SQL 中的 where 子句。Filter 包含如下类型。

1. Selector Filter

Selector Filter 的功能类似于 SQL 中的 `where key=value`。Selector Filter 的 JSON 示例如下：

```
"filter": { "type": "selector", "dimension": <dimension_string>,  
  "value": <dimension_value_string> }
```

2. Regex Filter

Regex Filter 允许用户用正则表达式来筛选维度，任何标准的 Java 支持的正则表达式 Druid 都支持。Regex Filter 的 JSON 示例如下：

```
"filter": { "type": "regex", "dimension": <dimension_string>, "pattern": <pattern_string> }
```

3. Logical Expression Filter

Logical Expression Filter 包含 `and`、`or` 和 `not` 三种过滤器。每一种都支持嵌套，可以构建丰富的逻辑表达式，并与 SQL 中的 `and`、`or` 和 `not` 相似。JSON 表达式示例如下：

```
"filter": { "type": "and", "fields": [<filter>, <filter>, ...] }  
"filter": { "type": "or", "fields": [<filter>, <filter>, ...] }  
"filter": { "type": "not", "field": <filter> }
```

4. Search Filter

Search Filter 通过字符串匹配过滤维度，支持多种匹配方式。JSON 示例如下：

```
{  
  "filter": {  
    "dimension": "product",  
    "query": {  
      "type": "insensitive_contains",  
      "value": "foo"  
    },  
    "type": "search"  
  }  
}
```

其中，`query` 中不同的 `type` 代表不同的匹配方式。

5. In Filter

In Filter 类似于 SQL 中的 `in`: `WHERE outlaw IN ('Good', 'Bad', 'Ugly')`。JSON 示例如下：

```
{
  "type": "in",
  "dimension": "outlaw",
  "values": ["Good", "Bad", "Ugly"]
}
```

6. Bound Filter

Bound Filter 其实就是比较过滤器，包含“大于”、“小于”和“等于”三种算子。Bound Filter 支持字符串比较，而且默认就是字符串比较，并基于字典序。如果要使用数字比较，则需要在查询中设定 `alphaNumeric` 的值为 `true`。需要注意的是，Bound Filter 默认的大小比较为“>=”或“<=”，因此如果要使用“<”或“>”，则需要指定 `lowerStrict` 的值为 `true` 或 `upperStrict` 的值为 `true`。具体的 JSON 表达式示例如下。

21 <= age <= 31:

```
{
  "type": "bound",
  "dimension": "age",
  "lower": "21", #默认为大于或等于21
  "upper": "31", #默认为小于或等于31
  "alphaNumeric": true #数字比较时指定alphaNumeric 为true
}
```

21 < age < 31:

```
{
  "type": "bound",
  "dimension": "age",
  "lower": "21",
  "lowerStrict": true, #指定lowerStrict为true，表示大于21
  "upper": "31",
  "upperStrict": true, #指定upperStrict为true，表示小于31
  "alphaNumeric": true
}
```

age < 31:

```
{
  "type": "bound",
  "dimension": "age",
```

```

"upper": "31",
"upperStrict": true,
"alphaNumeric": true
}

foo <= name <= hoo:
{
  "type": "bound",
  "dimension": "name",
  "lower": "foo",
  "upper": "hoo"
}

```

7. JavaScript Filter

如果上述 Filter 不能满足要求, Druid 还可以通过自己写 JavaScript Filter 来过滤维度,但是只支持一个入参,就是 Filter 里指定的维度的值,返回 true 或 false。JSON 表达式示例如下:

```

{
  "type": "javascript",
  "dimension": <dimension_string>,
  "function": "function(value) { <...> }"
}

```

例如 `foo <= name <= hoo`:

```

{
  "type": "javascript",
  "dimension": "name",
  "function": "function(x) { return(x >= 'foo' && x <= 'hoo') }"
}

```

6.2.2 Aggregator

Aggregator, 即聚合器, 若在摄入阶段就指定, 则会在 roll up 时就进行计算; 当然, 也

可以在查询时指定。聚合器包含如下详细类型。

1. Count Aggregator

Count Aggregator 计算 Druid 的数据行数，而 Count 就是反映被聚合的数据的计数。如果查询 Roll up 后有多少条数据，查询语句 JSON 示例如下：

```
{ "type" : "count", "name" : <output_name> }
```

如果要查询摄入了多少条原始数据，在查询时使用 longSum，JSON 示例如下：

```
{ "type" : "longSum", "name" : <output_name>, "fieldName" : "count" }
```

2. Sum Aggregator

第一类是 longSum Aggregator，它负责 64 位有符号整型的求和。JSON 示例如下：

```
{ "type" : "longSum", "name" : <output_name>, "fieldName" : <metric_name> }
```

第二类是 doubleSum Aggregator，它负责 64 位浮点数的求和。JSON 示例如下：

```
{ "type" : "doubleSum", "name" : <output_name>, "fieldName" : <metric_name> }
```

3. Min / Max Aggregator

第一类是 doubleMin Aggregator，它负责计算指定 Metric 的值和 Double.POSITIVE_INFINITY 的最小值。

第二类是 doubleMax Aggregator，它负责计算指定 Metric 的值和 Double.NEGATIVE_INFINITY 的最大值。

第三类是 longMin Aggregator，它负责计算指定 Metric 的值和 Long.MAX_VALUE 的最小值。

第四类是 longMax Aggregator 它负责计算指定 Metric 的值和 Long.MIN_VALUE 的最大值。

上述几类聚合器的 JSON 都比较相似，基本如下：

```
{ "type" : "doubleMin", "name" : <output_name>, "fieldName" : <metric_name> }  
{ "type" : "doubleMax", "name" : <output_name>, "fieldName" : <metric_name> }  
{ "type" : "longMin", "name" : <output_name>, "fieldName" : <metric_name> }  
{ "type" : "longMax", "name" : <output_name>, "fieldName" : <metric_name> }
```

4. Cardinality Aggregator

在查询时, Cardinality Aggregator 使用 HyperLogLog 算法计算给定维度集合的基数。需要注意的是, Cardinality Aggregator 比 HyperUnique Aggregator 要慢很多, 因为 HyperUnique Aggregator 在摄入阶段就会为 Metric 做聚合, 因此在通常情况下, 对于单个维度求基数, 比较推荐使用 HyperUnique Aggregator。

JSON 示例如下:

```
{
  "type": "cardinality",
  "name": "<output_name>",
  "fieldNames": [ <dimension1>, <dimension2>, ... ],
  "byRow": <false | true> # (optional, defaults to false)
}
```

byRow 为 false 时, 类似于以下 SQL:

```
SELECT COUNT(DISTINCT(value)) FROM (
  SELECT dim_1 as value FROM <datasource>
  UNION
  SELECT dim_2 as value FROM <datasource>
  UNION
  SELECT dim_3 as value FROM <datasource>
)
```

byRow 为 true 时, 类似于以下 SQL:

```
SELECT COUNT(*) FROM ( SELECT DIM1, DIM2, DIM3 FROM <datasource> GROUP BY DIM1, DIM2,
  DIM3 )
```

5. HyperUnique Aggregator

HyperUnique Aggregator 使用 HyperLogLog 算法计算指定维度的基数。在摄入阶段指定 Metric, 从而在查询时使用。JSON 示例如下:

```
{ "type": "hyperUnique", "name": <output_name>, "fieldName": <metric_name> }
```

除了 HyperLogLog, 近似的算法 Druid 还使用到了 ThetaSketch, 在第7章“高级功能和特性”中会有详细介绍。

6. Filtered Aggregator

Filtered Aggregator 可以在 aggregation 中指定 Filter 规则。只对满足规则的维度进行聚合，以提升聚合效率。JSON 示例如下：

```
{
  "type": "filtered",
  "filter": {
    "type": "selector",
    "dimension": <dimension>,
    "value": <dimension value>
  }
  "aggregator": <aggregation>
}
```

其中 <aggregator> 部分的拼写参照其他 Aggregator 的规则。

7. JavaScript Aggregator

如果上述聚合器无法满足需求，Druid 还提供了 JavaScript Aggregator。用户可以自己写 JavaScript function，其中指定的列即为 function 的入参。但是 JavaScript Aggregator 的执行性能要比本地 Java Aggregator 慢很多。因此，如果要追求性能，就需要自己实现本地 Java Aggregator。JavaScript Aggregator 的 JSON 示例如下：

```
{
  "type": "javascript",
  "name": "<output_name>",
  "fieldNames": [ <column1>, <column2>, ... ],
  "fnAggregate": "function(current, column1, column2, ...) {
    <updates partial aggregate (current) based on the current row
    values>
    return <updated partial aggregate>
  }",
  "fnCombine": "function(partialA, partialB) { return <combined partial results>; }",
  "fnReset": "function() { return <initial value>; }"
}
```

例子如下：

```
{
  "type": "javascript",
```

```

"name": "sum(log(x)*y) + 10",
"fieldNames": ["x", "y"],
"fnAggregate": "function(current, a, b) { return current + (Math.log(a) * b); }",
"fnCombine"   : "function(partialA, partialB) { return partialA + partialB; }",
"fnReset"     : "function() { return 10; }"
}

```

6.2.3 Post-Aggregator

Post-Aggregator 可以对 Aggregator 的结果进行二次加工并输出。最终的输出既包含 Aggregation 的结果，也包含 Post-Aggregator 的结果。如果使用 Post-Aggregator，则必须包含 Aggregator。Post-Aggregator 包含如下类型。

1. Arithmetic Post-Aggregator

Arithmetic Post-Aggregator 支持对 Aggregator 的结果和其他 Arithmetic Post-Aggregator 的结果进行加“+”、减“-”、乘“*”、除“/”和“quotient”计算。需要注意的是：

- 对于“/”，如果分母为0，则返回0。
- “quotient”不判断分母是否为0。
- 当 Arithmetic Post-Aggregator 的结果参与排序时，默认使用 float 类型。用户可以手动通过 ordering 字段指定排序方式。

JSON 示例如下：

```

"postAggregation" : {
  "type" : "arithmetic",
  "name" : <output_name>,
  "fn"   : <arithmetic_function>,
  "fields": [<post_aggregator>, <post_aggregator>, ...],
  "ordering" : <null (default), or "numericFirst">
}

```

2. Field Accessor Post-Aggregator

Field Accessor Post-Aggregator 返回指定的 Aggregator 的值，在 Post-Aggregator 中大部分情况下使用 fieldAccess 来访问 Aggregator。在 fieldName 中指定 Aggregator 里定义的名称，如果对 HyperUnique 的结果进行访问，则需要使用 hyperUniqueCardinality。Field Accessor Post-Aggregator 的 JSON 示例如下：

```
{
  "type": "fieldAccess",
  "name": <output_name>,
  "fieldName": <aggregator_name>
}
```

3. Constant Post-Aggregator

Constant Post-Aggregator 会返回一个常数，比如 100。可以将 Aggregator 返回的结果转换为百分比。JSON 示例如下：

```
{
  "type": "constant",
  "name": <output_name>,
  "value": <numerical_value>
}
```

4. HyperUnique Cardinality Post-Aggregator

HyperUnique Cardinality Post-Aggregator 得到 HyperUnique Aggregator 的结果，使之能参与到 Post-Aggregator 的计算中。JSON 示例如下：

```
{
  "type": "hyperUniqueCardinality",
  "name": <output name>,
  "fieldName": <the name field value of the hyperUnique aggregator>
}
```

例子如下：

```
{
  ...
  "aggregations": [
    {
      "name": "rows",
      "type": "count"
    },
    {
      "fieldName": "uniques",
      "name": "unique_users",
      "type": "hyperUnique"
    }
  ]
}
```

```

    },
    "postAggregations": [
      {
        "fields": [
          {
            "fieldName": "unique_users",
            "type": "hyperUniqueCardinality"
          },
          {
            "fieldName": "rows",
            "name": "rows",
            "type": "fieldAccess"
          }
        ],
        "fn": "/",
        "name": "average_users_per_row",
        "type": "arithmetic"
      }
    ]
  },
  ...
}

```

Post-Aggregator 的计算可以嵌套，以此得到更加丰富的计算方式。一个简单的示例如下：

```

{
  "aggregations": [
    {
      "name": "rows",
      "type": "count"
    },
    {
      "fieldName": "total",
      "name": "tot",
      "type": "doubleSum"
    }
  ],
  "postAggregations": [

```

```

{
  "fields": [
    {
      "fields": [
        {
          "fieldName": "tot",
          "name": "tot",
          "type": "fieldAccess"
        },
        {
          "fieldName": "rows",
          "name": "rows",
          "type": "fieldAccess"
        }
      ],
      "fn": "/",
      "name": "div",
      "type": "arithmetic"
    },
    {
      "name": "const",
      "type": "constant",
      "value": 100
    }
  ],
  "fn": "*",
  "name": "average",
  "type": "arithmetic"
}
...
}

```

更多高级聚合器如“近似直方图”“DataSketch”等内容见第7章。

6.2.4 Search Query

Search Query 在 Filter 的 search 和 search 查询中都会用到。Search Query 定义了如下几种字符串匹配方式。

1. contains

如果指定的维度的值包含给定的字符串，则匹配。contains 可以通过 case_sensitive 指定是否区分大小写。JSON 示例如下：

```
{
  "type" : "contains",
  "case_sensitive" : true,
  "value" : "some_value"
}
```

2. insensitive_contains

如果指定的维度的值包含给定的字符串，则匹配。不区分大小写，没有 case_sensitive 字段。如果 contains 中的 case_sensitive 设置为 false，则和 insensitive_contains 等价。insensitive_contains 的 JSON 示例如下：

```
{
  "type" : "insensitive_contains",
  "value" : "some_value"
}
```

3. fragment

如果指定的维度的值的任意部分包含给定的字符串，则匹配。JSON 示例如下：

```
{
  "type" : "fragment",
  "case_sensitive" : false,
  "values" : ["fragment1", "fragment2"]
}
```

6.2.5 Interval

在查询中指定时间区间。Interval 中的时间是 ISO-8601 格式。对于中国用户，所在时区为东 8 区，因此需要在时间中加入“+08:00”。查询时间格式为：

"intervals" : ["2016-08-28T00:00:00+08:00/2016-08-29T00:00:00+08:00"]

需要注意的是，这里 intervals 的时间区间为前闭后开：starttime <= datetime < endtime。如果查询时需要将最后 1 秒包含在内，那么 endtime 最好往后推 1 秒。

6.2.6 Context

Context 可以在查询中指定一些参数。Context 并不是查询的必选项，因此在查询中不指定 Context 时，则会使用 Context 中的默认参数。Context 支持的字段如下：

字段名	默认值	描述
timeout	0（未超时）	查询超时时间，单位是毫秒
priority	0	查询优先级
queryId	自动生成	唯一标识一次查询的 id，可以用该 id 取消查询
useCache	true	此次查询是否利用查询缓存，如果手动指定，则会覆盖查询节点或历史节点配置的值
populateCache	true	此次查询的结果是否缓存，如果手动指定，则会覆盖查询节点或历史节点配置的值
bySegment	false	指定为 true 时，将在返回结果中显示关联的 Segment
finalize	true	是否返回 Aggregator 的最终结果，例如 HyperUnique，指定为 false 时，将返回序列化的结果，而不是估算的基数数值
chunkPeriod	0（off）	指定是否将长时间跨度的查询切分为多个短时间跨度进行查询，需要配置 druid.processing.numThreads 的值
minTopNThreshold	1000	配置每个 Segment 返回的 TopN 的数量用于合并，从而得到最终的 TopN
maxResults	500000	配置 GroupBy 最多能处理的结果集条数，默认值在历史节点的配置项 druid.query.groupBy.maxIntermediateRows 中指定，查询时该字段的值只能小于配置项的值
maxIntermediateRows	50000	指定一些查询参数，如结果是否进缓存
groupByIsSingle-Threaded	false	是否使用单线程执行 GroupBy，默认值在历史节点的配置项 druid.query.groupBy.singleThreaded 中指定

6.3 案例介绍

本章使用真实案例来介绍如何进行查询，供读者参考。

首先介绍本案例的相关背景。案例中的广告为一个拼有标识广告标识的 URL，如：

`http://www.mejia.wang/?ad_source=google&ad_campaign=test&ad_media=vedio`

系统会为每个访问该 URL 的用户生成一个 `user_id`。当打开广告页面后，上面会有一个发起聊天的按钮（命名为“接待组件”。在有些需求中可能是“购买”按钮或者是其他标志转化率的按钮或时间），每次访问和点击该按钮的操作都会被上报。如果发起聊天，就会进入该广告主的客户库。每次访问都会产生一条记录，如果在访问中点击了“接待组件”，则会在相应的地方填上。产生的字段如下：

字段名	描述
tid	id
timestamp	访问时间
corpuin	广告主 id
host	域名
device_type	用户访问设备类型: 1.PC 2.Mobile 3.other
is_new	该用户是否是新访客
ad_source	广告来源
ad_media	广告媒介
ad_campaign	广告系列
user_id	用户 id
click_user_id	如果用户点击了接待组件，则将 user_id 填上
new_user_id	如果用户是新用户，则将 user_id 这里也填上

接下来，我们结合本案例来介绍 Druid 的查询语法。

案例中所需的 Data Schema 定义如下：

```
{
  "spec": {
    "dataSchema": {
      "dataSource": "visitor_statistics",
```

```

"granularitySpec": {
  "intervals": [
    "2016-08-28T00:00:00+08:00/2016-08-29T00:00:00+08:00"
  ],
  "queryGranularity": "day",
  "segmentGranularity": "day",
  "type": "uniform"
},
"metricsSpec": [
  {
    "name": "count",
    "type": "count"
  },
  {
    "fieldName": "user_id",
    "name": "visit_count",
    "type": "hyperUnique"
  },
  {
    "fieldName": "new_user_id",
    "name": "new_visit_count",
    "type": "hyperUnique"
  },
  {
    "fieldName": "click_userid",
    "name": "click_visit_count",
    "type": "hyperUnique"
  }
],
"parser": {
  "parseSpec": {
    "dimensionsSpec": {
      "dimensionExclusions": [],
      "dimensions": [
        "tid",
        "corpuin",
        "host",
        "device_type",

```

```

        "is_new",
        "ad_source",
        "ad_media",
        "ad_campaign"
    ],
    "format": "json",
    "timestampSpec": {
        "column": "timestamp",
        "format": "auto"
    }
},
"type": "hadoopString"
},
"ioConfig": {
    "inputSpec": {
        "paths": "hdfs://${集群地址}/olap/visitor_stat/",
        "type": "static"
    },
    "type": "hadoop"
},
"tuningConfig": {
    "cleanupOnFailure": false,
    "maxRowsInMemory": 100000,
    "partitionsSpec": {
        "targetPartitionSize": 5000000,
        "type": "hashed"
    },
    "type": "hadoop"
}
},
"type": "index_hadoop"
}

```

这里使用 HyperUnique 进行访客数统计，对 user_id、new_user_id 和 click_userid 进行 HyperUnique 摄入聚合，这样就可以方便、快捷地获取各种行为对应的访客数。

6.4 Timeseries

对于需要统计一段时间内的汇总数据，或者是指定时间粒度的汇总数据，Druid 通过 Timeseries 来完成。例如，对指定客户 id 和 host，统计一段时间内的访问次数、访客数、新访客数、点击按钮数、新访客比率与点击按钮比率，我们可以用如下查询语句。

```
{
  "queryType": "timeseries",
  "dataSource": "visitor_statistics",
  "granularity": "all",
  "filter": {
    "type": "and",
    "fields": [
      {
        "type": "selector",
        "dimension": "host",
        "value": "www.mejia.wang"
      },
      {
        "type": "selector",
        "dimension": "corpuin",
        "value": "2852199351"
      }
    ]
  },
  "aggregations": [
    {
      "type": "longSum",
      "name": "pv",
      "fieldName": "count"
    },
    {
      "type": "hyperUnique",
      "name": "visitor_count",
      "fieldName": "visit_count"
    },
    {
      "type": "hyperUnique",
      "name": "new_visitor_ratio",
      "fieldName": "new_visitor_ratio"
    },
    {
      "type": "hyperUnique",
      "name": "click_button_ratio",
      "fieldName": "click_button_ratio"
    }
  ]
}
```

```

    "name": "new_visitor_count",
    "fieldName": "new_visit_count"
  },
  {
    "type": "hyperUnique",
    "name": "click_visitor_count",
    "fieldName": "click_visit_count"
  }
],

```

```

"postAggregations": [
  {

```

```

    "type": "arithmetic",
    "name": "new_visitor_rate",
    "fn": "/",
    "fields": [
      {
        "type": "hyperUniqueCardinality",
        "fieldName": "new_visitor_count"
      },
      {
        "type": "hyperUniqueCardinality",
        "fieldName": "visitor_count"
      }
    ]
  },
  {
    "type": "arithmetic",
    "name": "click_rate",
    "fn": "/",
    "fields": [
      {
        "type": "hyperUniqueCardinality",
        "fieldName": "click_visitor_count"
      },
      {
        "type": "hyperUniqueCardinality",
        "fieldName": "visitor_count"
      }
    ]
  }
]

```

```
    ]
  },
  "intervals": [
    "2016-08-07T00:00:00+08:00/2016-09-05T23:59:59+08:00"
  ]
}
```

Timeseries 查询包含如下部分。

字段名	描述	是否必需
queryType	对于 Timeseries 查询，该字段的值必须是 Timeseries	是
dataSource	要查询数据集 dataSource 名字	是
intervals	查询时间区间范围，ISO-8601 格式	是
granularity	查询结果进行聚合的时间粒度	是
filter	过滤器	否
aggregations	聚合器	是
postAggregations	后聚合器	否
descending	是否降序	否
context	指定一些查询参数，如结果是否进缓存等	否

Timeseries 输出每个时间粒度内指定条件的统计信息，通过 filter 指定过滤条件，通过 aggregations 和 postAggregations 指定聚合方式。

Timeseries 不能输出维度信息，granularity 支持 all, none, second, minute, fifteen_minute, thirty_minute, hour, day, week, month, quarter, year。

- all，汇总为 1 条输出。
- none，不被推荐使用。
- 其他的，则输出相应粒度的统计信息。

输出可能如下：

```
[
  {
    "timestamp": "2016-08-27T16:00:00.000Z",
    "result": {
      "pv": 30.000000061295435,
      "visit_count": 5.006113467958146,
      "new_visitor_count": 0,
      "click_visitor_count": 5.006113467958146,
      "new_visitor_rate": 0,
      "click_rate": 1,
    }
  }
]
```

Timeseries 查询默认会给没有数据的 buckets 填 0，例如 granularity 设置为 day，查询 2012-01-01 到 2012-01-03 的数据，但是如果 2012-01-02 没有数据，会收到如下结果：

```
[
  {
    "timestamp": "2012-01-01T00:00:00.000Z",
    "result": { "sample_name1": <some_value> }
  },
  {
    "timestamp": "2012-01-02T00:00:00.000Z",
    "result": { "sample_name1": 0 }
  },
  {
    "timestamp": "2012-01-03T00:00:00.000Z",
    "result": { "sample_name1": <some_value> }
  }
]
```

如果不希望 Druid 自动补 0，可以在请求的 context 中指定 skipEmptyBuckets 的值为 true，例子如下：

```
{
  "queryType": "timeseries",
  "dataSource": "sample_datasource",
```



```

"granularity": "day",
"aggregations": [
  { "type": "longSum", "name": "sample_name1", "fieldName": "sample_fieldName1" }
],
"intervals": [ "2012-01-01T00:00:00.000/2012-01-04T00:00:00.000" ],
"context" : {
  "skipEmptyBuckets": "true"
}
}

```

但是需要注意的是，如果 2012-01-02 对应的 Segment 不存在，即使不设置 skipEmptyBuckets 为 true，Druid 也不会补 0。

6.5 TopN

TopN 是非常常见的查询类型，返回指定维度和排序字段的有序 top-n 序列。TopN 支持返回前 N 条记录，并支持指定 Metric 为排序依据。例如，对指定广告主 id=2852199100 和指定 host=www.mejia.wang，以及来自 PC 或手机访问，希望获取访客数最高的 3 个 ad_source，以及每个 ad_source 对应的访问次数、访客数、新访客数、点击按钮数、新访客比率、点击按钮比率、ad_campaign 与 ad_media 的组合个数，查询示例如下：

```

{
  "queryType": "topN",
  "dataSource": "visitor_statistics",
  "granularity": "all",
  "dimension": "ad_source",
  "threshold": 3,
  "metric": {
    "type": "numeric",
    "metric": "pv"
  },
  "filter": {
    "type": "and",
    "fields": [
      {
        "type": "selector",
        "dimension": "host",
        "value": "www.mejia.wang"
      }
    ]
  }
}

```

```

    },
    {
      "type": "selector",
      "dimension": "corpuin",
      "value": "2852199100"
    },
    {
      "type": "or",
      "fields": [
        {
          "type": "selector",
          "dimension": "device_type",
          "value": "1"
        },
        {
          "type": "selector",
          "dimension": "device_type",
          "value": "2"
        }
      ]
    },
    "aggregations": [
      {
        "type": "longSum",
        "name": "pv",
        "fieldName": "count"
      },
      {
        "type": "hyperUnique",
        "name": "visitor_count",
        "fieldName": "visit_count"
      },
      {
        "type": "hyperUnique",
        "name": "new_visitor_count",
        "fieldName": "new_visit_count"
      }
    ]
  }
}

```

```

    },
    {
      "type": "hyperUnique",
      "name": "click_visitor_count",
      "fieldName": "click_visit_count"
    },
    {
      "type": "cardinality",
      "name": "sub_count",
      "fieldNames": [
        "ad_campaign",
        "ad_media"
      ],
      "byRow": true
    }
  ],
  "postAggregations": [
    {
      "type": "arithmetic",
      "name": "new_visitor_rate",
      "fn": "/",
      "fields": [
        {
          "type": "hyperUniqueCardinality",
          "fieldName": "new_visitor_count"
        },
        {
          "type": "hyperUniqueCardinality",
          "fieldName": "visitor_count"
        }
      ]
    },
    {
      "type": "arithmetic",
      "name": "click_rate",
      "fn": "/",
      "fields": [
        {

```

```
        "type": "hyperUniqueCardinality",
        "fieldName": "click_visitor_count"
    },
    {
        "type": "hyperUniqueCardinality",
        "fieldName": "visitor_count"
    }
]
},
"intervals": [
    "2016-08-30T00:00:00+08:00/2016-09-05T23:59:59+08:00"
]
}
```

TopN 查询包含如下部分。

字段名	描述	是否必需
queryType	对于 TopN 查询，该字段的值必须是 topN	是
dataSource	要查询数据集 dataSource 名字	是
intervals	查询时间区间范围，ISO-8601 格式	是
granularity	查询结果进行聚合的时间粒度	是
filter	过滤器	否
aggregations	聚合器	是
postAggregations	后聚合器	否
dimension	进行 TopN 查询的维度，一个 TopN 查询指定且只能指定一个维度，如 URL	是
threshold	TopN 的 N 取值	是
metric	进行统计并排序的 Metric，如 PV	是
context	指定一些查询参数，如结果是否进缓存等	否

上述查询 JSON 基本上包含了 TopN 查询中能用到的所有特性。

- **filter**: 过滤指定的条件。支持 “and”, “or”, “not”, “in”, “regex”, “search”, “bound”。
- **aggregations**: 聚合器。用到的聚合函数和字段需要在 `metricsSpec` 中定义。HyperUnique 采用 HyperLogLog 近似对指定字段求基数, 这里用来算出各种行为的访客数。cardinality 用来计算指定维度的基数, 它与 HyperUnique 不同的是支持多个维度, 但是性能比 HyperUnique 差。
- **postAggregations**: 对 aggregations 的结果进行二次加工, 支持加、减、乘、除等运算。
- **metric**: TopN 专属, 指定排序依据。它有如下使用方式:

"metric": "<metric_name>" //默认方式, 升序排列

```
"metric": {
  "type": "numeric", //指定按照numeric 降序排列
  "metric": "<metric_name>"
}
```

```
"metric": {
  "type": "inverted", //指定按照numeric 升序排列
  "metric": <delegate_top_n_metric_spec>
}
```

```
"metric": {
  "type": "lexicographic", //指定按照字典序排序
  "previousStop": "<previousStop_value>" //如"b", 按照字典序, 排到 "b" 开头的为止
}
```

```
"metric": {
  "type": "alphaNumeric", //指定数字排序
  "previousStop": "<previousStop_value>"
}
```

查询结果如下:

```
[
  {
    "timestamp": "2016-08-29T16:00:00.000Z",
```

```

"result": [
  {
    "pv": 5,
    "new_visitor_rate": 1,
    "click_rate": 0,
    "sub_count": 1.0002442201269182,
    "new_visitor_count": 7.011990219885757,
    "visitor_count": 7.011990219885757,
    "ad_source": "baidu",
    "click_visitor_count": 0
  },
  {
    "pv": 4,
    "new_visitor_rate": 1,
    "click_rate": 0,
    "sub_count": 1.0002442201269182,
    "new_visitor_count": 4.003911343725148,
    "visitor_count": 4.003911343725148,
    "ad_source": "google",
    "click_visitor_count": 0
  },
  {
    "pv": 3,
    "new_visitor_rate": 1,
    "click_rate": 0,
    "sub_count": 1.0002442201269182,
    "new_visitor_count": 4.003911343725148,
    "visitor_count": 4.003911343725148,
    "ad_source": "sogou",
    "click_visitor_count": 0
  }
]
}
]

```

需要注意的是，topN 是一个近似算法，每一个 Segment 返回前 1000 条进行合并再得到最后的结果，如果 dimension 的基数在 1000 以内，则是准确的，超过 1000 就是近似值了。

6.6 GroupBy

GroupBy 类似于 SQL 中的 `group by` 操作，能对指定的多个维度进行分组，也支持对指定的维度进行排序，并输出 `limit` 行数。同时，支持 `having` 操作。GroupBy 与 TopN 相比，可以指定更多的维度，但性能比 TopN 要差很多。如果是对时间范围进行聚合，输出各个时间的统计数据，类似于 `group by hour` 之类的操作，通常应该使用 Timeseries。如果是对单个维度进行 `group by`，则应尽量使用 TopN。这两者的性能比 GroupBy 要好很多，在 Druid 0.9.2 版本中，对 GroupBy 有一个优化，可以通过在 Context 中指定使用新的算法。GroupBy 支持 `limit`，在 `limitSpec` 中按照指定 Metric 排序，不过不支持 `offset`。

例如，希望查询每组 `ad_source`、`ad_campaign` 和 `ad_media` 对应的访客数、新访客数、点击按钮数、新访客比率和点击按钮比率，查询示例如下：

```
{
  "queryType": "groupBy",
  "dataSource": "visitor_statistics",
  "granularity": "all",
  "dimensions": [
    "ad_source",
    "ad_campaign",
    "ad_media"
  ],
  "limitSpec": {
    "type": "default",
    "limit": 1000,
    "columns": [
      {
        "dimension": "visitor_count",
        "direction": "descending"
      }
    ]
  },
  "filter": {
    "type": "and",
    "fields": [
      {
        "type": "selector",
        "dimension": "host",
```

```

        "value": "www.mejia.wang"
    },
    {
        "type": "selector",
        "dimension": "is_ad",
        "value": 1
    },
    {
        "type": "selector",
        "dimension": "corpuin",
        "value": "2852199351"
    },
    {
        "type": "or",
        "fields": [
            {
                "type": "selector",
                "dimension": "device_type",
                "value": "1"
            },
            {
                "type": "selector",
                "dimension": "device_type",
                "value": "2"
            }
        ]
    }
],
},
"aggregations": [
    {
        "type": "hyperUnique",
        "name": "visitor_count",
        "fieldName": "visit_count"
    },
    {
        "type": "hyperUnique",
        "name": "new_visitor_count",

```



```

        "fieldName": "new_visit_count"
    },
    {
        "type": "hyperUnique",
        "name": "click_visitor_count",
        "fieldName": "click_count"
    }
],
"postAggregations": [
    {
        "type": "arithmetic",
        "name": "new_visitor_rate",
        "fn": "/",
        "fields": [
            {
                "type": "hyperUniqueCardinality",
                "fieldName": "new_visitor_count"
            },
            {
                "type": "hyperUniqueCardinality",
                "fieldName": "visitor_count"
            }
        ]
    },
    {
        "type": "arithmetic",
        "name": "click_rate",
        "fn": "/",
        "fields": [
            {
                "type": "hyperUniqueCardinality",
                "fieldName": "click_visitor_count"
            },
            {
                "type": "hyperUniqueCardinality",
                "fieldName": "visitor_count"
            }
        ]
    }
]

```

```
    }
  ],
  "intervals": [
    "2016-08-29T00:00:00+08:00/2016-09-04T23:59:59+08:00"
  ]
}
```

GroupBy 查询包含如下部分。

字段名	描述	是否必需
queryType	对于 GroupBy 查询，该字段的值必须是 groupBy	是
dataSource	要查询数据集 dataSource 名字	是
dimensions	进行 GroupBy 查询的维度集合	是
limitSpec	对统计结果进行排序，取 limit 的行数	否
having	对统计结果进行筛选	否
granularity	查询结果进行聚合的时间粒度	是
filter	过滤器	否
aggregations	聚合器	是
postAggregations	后聚合器	否
intervals	查询时间区间范围，ISO-8601 格式	是
context	指定一些查询参数，如结果是否进缓存等	否

GroupBy 特有的字段为 limitSpec 和 having。

(1) limitSpec

指定排序规则和 limit 的行数。JSON 示例如下：

```
{
  "type": "default",
  "limit": <integer_value>,
  "columns": [list of OrderByColumnSpec],
}
```

其中 columns 是一个数组，可以指定多个排序字段，排序字段可以是 demension 或 metric，

指定排序规则的拼写方式:

```
{
  "dimension": "<Any dimension or metric name>",
  "direction": "<\"ascending\"|\"descending\">"
}
```

示例如下:

```
"limitSpec": {
```

"type": "default",	类型	字符串
"limit": 1000,		
"columns": [
{		
"dimension": "visitor_count",		
"direction": "descending"		
},		
{		
"dimension": "click_visitor_count",		
"direction": "ascending"		
}		
]		
}		

(2) having

类似于 SQL 中的 having 操作, 对 GroupBy 的结果进行筛选。支持大于、等于、小于、selector、and、or 和 not 等操作。JSON 示例如下:

```
{
  "type": "greaterThan",
  "aggregation": "<aggregate_metric>",
  "value": <numeric_value>
}
```

```
{
  "type": "equalTo",
  "aggregation": "<aggregate_metric>",
  "value": <numeric_value>
}
```

```

{
  "type": "lessThan",
  "aggregation": "<aggregate_metric>",
  "value": <numeric_value>
}

{
  "type": "dimSelector",
  "dimension": "<dimension>",
  "value": <dimension_value>
}

{
  "type": "and",
  "havingSpecs": [<having clause>, <having clause>, ...]
}

{
  "type": "or",
  "havingSpecs": [<having clause>, <having clause>, ...]
}

{
  "type": "not",
  "havingSpec": <having clause>
}

```

在最新发布的 0.9.2 版本中，GroupBy 可以在 context 中指定使用新算法，指定方式为：

```
"context": { "groupByStrategy": "v2" }
```

如果不指定，默认使用 v1。

6.7 Select

Select 类似于 SQL 中的 select 操作，Select 用来查看 Druid 中存储的数据，并支持按照指定过滤器和时间段查看指定维度和 Metric。能通过 descending 字段指定排序顺序，并支持分页拉取，但不支持 aggregations 和 postAggregations。

```

{
  "dataSource": "visitor_statistics",
  "descending": "false",
  "dimensions": [
    "tid",
    "corpuin",
    "host",
    "device_type",
    "is_new",
    "ad_source",
    "ad_media",
    "ad_campaign"
  ],
  "intervals": [
    "2016-08-29/2016-08-31"
  ],
  "metrics": [
    "count",
    "visit_count",
    "new_visit_count",
    "click_visit_count"
  ],
  "pagingSpec": {
    "pagingIdentifiers": {},
    "threshold": 5
  },
  "queryType": "select"
}

```

在 `pagingSpec` 中指定分页拉取的 `offset` 和条目数，在结果中会返回下次拉取的 `offset`。JSON 示例如下：

```

{
  ...
  "pagingSpec": {"pagingIdentifiers": {}, "threshold": 5, "fromNext": true}
}

```

6.8 Search

Search 查询返回匹配中的维度，类似于 SQL 中的 like 操作，但是支持更多的匹配操作。

```
{
  "type" : "insensitive_contains",
  "value" : "some_value"
}

{
  "type" : "fragment",
  "case_sensitive" : false,
  "values" : ["fragment1", "fragment2"]
}

{
  "type" : "contains",
  "case_sensitive" : true,
  "value" : "some_value"
}

{
  "type" : "regex",
  "pattern" : "some_pattern"
}
```

一个 Search 查询的 JSON 示例如下：

```
{
  "dataSource": "sample_datasource",
  "granularity": "day",
  "intervals": [
    "2013-01-01T00:00:00.000/2013-01-03T00:00:00.000"
  ],
  "query": {
    "type": "insensitive_contains",
    "value": "Ke"
  },
  "queryType": "search",
  "searchDimensions": [
    "dim1",
    "dim2"
  ]
}
```

```

    ],
    "sort": {
      "type": "lexicographic"
    }
  }
}

```

返回结果如下:

```

[
  {
    "result": [
      {
        "count": 3,
        "dimension": "dim1",
        "value": "Ke$ha"
      },
      {
        "count": 1,
        "dimension": "dim2",
        "value": "Ke$haForPresident"
      }
    ],
    "timestamp": "2012-01-01T00:00:00.000Z"
  },
  {
    "result": [
      {
        "count": 1,
        "dimension": "dim1",
        "value": "SomethingThatContainsKe"
      },
      {
        "count": 2,
        "dimension": "dim2",
        "value": "SomethingElseThatContainsKe"
      }
    ],
    "timestamp": "2012-01-02T00:00:00.000Z"
  }
]

```

需要注意的是, Search 只是返回匹配中维度, 不支持其他聚合操作。如果要将 Search 作为查询条件进行 TopN、GroupBy 或 Timeseries 等操作, 则可以在 filter 字段中指定各种过滤方式。filter 字段也支持正则匹配。

6.9 元数据查询

Druid 支持对 DataSource 的基础元数据进行查询。可以通过 timeBoundary 查询 DataSource 的最早和最晚的时间点; 通过 segmentMetadata 查询 Segment 的元信息, 如有哪些 column、metric、aggregator 和查询粒度等信息; 通过 dataSourceMetadata 查询 DataSource 的最近一次摄入数据的时间戳。查询 JSON 示例分别如下:

1. timeBoundary

```
{
  "queryType": "timeBoundary",
  "dataSource": "sample_datasource",
  "bound": "<\"maxTime\" | \"minTime\">"
}
```

返回结果如下:

```
[
  {
    "result": {
      "maxTime": "2013-05-09T18:37:00.000Z",
      "minTime": "2013-05-09T18:24:00.000Z"
    },
    "timestamp": "2013-05-09T18:24:00.000Z"
  }
]
```

2. segmentMetadata

```
{
  "queryType": "segmentMetadata",
  "dataSource": "sample_datasource",
  "intervals": ["2013-01-01/2014-01-01"]
}
```


返回结果如下:

```
{
  "aggregators": {
    "metric1": {
      "fieldName": "metric1",
      "name": "metric1",
      "type": "longSum"
    },
    "columns": {
      "__time": {
        "cardinality": null,
        "errorMessage": null,
        "hasMultipleValues": false,
        "size": 407240380,
        "type": "LONG"
      },
      "dim1": {
        "cardinality": 1944,
        "errorMessage": null,
        "hasMultipleValues": false,
        "size": 100000,
        "type": "STRING"
      },
      "dim2": {
        "cardinality": 1504,
        "errorMessage": null,
        "hasMultipleValues": true,
        "size": 100000,
        "type": "STRING"
      },
      "metric1": {
        "cardinality": null,
        "errorMessage": null,
        "hasMultipleValues": false,
        "size": 100000,

```

```
        "type": "FLOAT"
    },
    "id": "some_id",
    "intervals": [
        "2013-05-13T00:00:00.000Z/2013-05-14T00:00:00.000Z"
    ],
    "numRows": 5000000,
    "queryGranularity": {
        "type": "none"
    },
    "size": 300000
}
]
```

segmentMetadata 支持更多的查询字段，不过这些字段都不是必需的，简介如下：

字段名	描述	是否必需
toInclude	可以指定哪些 column 在返回结果中呈现，可以填 all, none, list	否
merge	将多个 Segment 的元信息合并到一个返回结果中	否
analysisTypes	指定返回 column 的哪些属性，如 size, intervals 等	是
lenientAggregatorMerge	true 或 false，设置为 true 时，将不同的 aggregator 合并显示	否
context	查询 Context，可以指定是否缓存查询结果等	否

toInclude 的使用方式如下：

```
"toInclude": { "type": "all" }
"toInclude": { "type": "none" }
"toInclude": { "type": "list", "columns": [<string list of column names>] }
```

analysisTypes 支持指定的属性：cardinality, minmax, size, intervals, queryGranularity, aggregators。

3. dataSourceMetadata

```
{  
  "queryType": "dataSourceMetadata",  
  "dataSource": "sample_datasource"  
}
```

返回结果如下:

```
[[  
  "timestamp": "2013-05-09T18:24:00.000Z",  
  "result": {  
    "maxIngestedEventTime": "2013-05-09T18:24:09.007Z",  
  }  
]]
```

6.10 小结

Druid 提供的查询方式非常丰富, 几乎涵盖了 OLAP 查询的方方面面, 并且很多查询语句与 SQL 几乎一致, 同时还支持开发者编写自己的聚合器。这些丰富的功能使得用户可以轻松、灵活地在 Druid 平台上进行数据探索。

Column	Segment	Annotation
Column 1	Segment 1	Annotation 1
Column 2	Segment 2	Annotation 2
Column 3	Segment 3	Annotation 3
Column 4	Segment 4	Annotation 4
Column 5	Segment 5	Annotation 5
Column 6	Segment 6	Annotation 6
Column 7	Segment 7	Annotation 7
Column 8	Segment 8	Annotation 8
Column 9	Segment 9	Annotation 9
Column 10	Segment 10	Annotation 10

第7章

高级功能和特性

本章介绍 Druid 的一些高级功能和特性，其中有些功能在开发过程中属于实验功能（Experimental Feature），有些功能尚未完成全面测试，但是其核心功能从实践效果来说还都非常稳定，可以放心使用。下面功能都是 Druid 未来发展的重要分析功能，是很多应用场景急需的一些功能。

- 近似直方图和分位数

- 预估数据（DataSketch）

- 地理索引和查询

- 路由器（Router）

- Kafka 索引服务

为了使用这些功能，有时候需要在配置文件中设置，让 Druid 装载这些模块。例如对于直方图模块，设置如下：

```
druid.extensions.loadList=["druid-histogram"]
```

该设置需要应用在索引节点和查询节点的配置文件中。

7.1 近似直方图（Approximate Histogram）

7.1.1 分位数和直方图

分位数（Quantile）是指将一组数据按照一定方法分为几个等份的数值点，分析其数据变量的趋势。常用的有中位数、四分位数、百分位数等。中位数（Median）是一个统计学专有名词，表示将数据按照大小排序，找到中间位置的那个数据。许多数据分析场景都需要使用百分位数（Percentile）的指标统计，例如，有些网络服务的响应时间都需满足：保证 99% 的访问响应时间小于 1 秒等。

直方图（Histogram）是一种统计报告图，由一系列高度不等的纵向条纹或线段表示数据分布的情况。一般用横轴表示数据类型，用纵轴表示分布情况。

7.1.2 实现原理

分位数的计算通常涉及对原始数据进行排序，排序本身是比较容易并行化的，但是这种方案需要将原始值都记录下来。Druid 的核心设计是通过预先聚合一些数据，大大提高访问性能。如果记录原始数据，将会极大地失去性能优势。

为了解决这个问题，Druid 采用一种近似的计算方式来获得中位数、直方图等。使用者可以在性能和准确度之间进行取舍和平衡，该方案也支持百亿级以上的数据规模。请参考 <http://jmlr.org/papers/volume11/ben-haim10a/ben-haim10a.pdf>。

由于每一时间段的数据都已经聚合，如果像图 7-1 一样保存原始数据，则几乎是不可能的，因为数据存储和处理都会变成巨大的问题。

timestamp	publisher	advertiser	gender	country	impressions	clicks	prices
2011-01-01T01:00:00Z	ultratrimfast.com	google.com	Male	USA	1800	25	[0.64, 1.93, 0.93, ...]
2011-01-01T01:00:00Z	bieberfever.com	google.com	Male	USA	2912	42	[0.65] 0.62, 0.45, ...]
2011-01-01T02:00:00Z	ultratrimfast.com	google.com	Male	UK	1953	17	[0.07, 0.34, 1.23, ...]
2011-01-01T02:00:00Z	bieberfever.com	google.com	Male	UK	3194	170	[0.53, 0.92, 0.12, ...]

图 7-1 保持原始数据的简单方案（数据量巨大）

因此，如何减少每一区段的数量，就是分位数设计的核心挑战。Druid 的实现方式是：对需要索引的列，保持固定数量的二元组，表示该数据的分布情况。固定数量意味着数据规模和计算时间可以保证。最初的想法是保存 < 数量, 中位数 > 二元组，但是这个二元组在进行累加计算时容易丢失信息。例如，两个中位数的和的含义不明显，累加越多，越

失去了中位数的意义。在 Druid 的最后实现中，采用的是 < 数量，重心 (centroid) > 作为二元组的实现。其中重心点的计算，是一种带权重的均值计算，有兴趣的同学，可以参考 <http://dl.acm.org/citation.cfm?id=1519389>。

数据格式如图 7-2 所示。

timestamp	publisher	advertiser	gender	country	impressions	clicks	AH_prices
2011-01-01T01:00:00Z	ultratrimfast.com	google.com	Male	USA	1800	25	[(1, .16), (48, .62), (83, .71), ...]
2011-01-01T01:00:00Z	bieberfever.com	google.com	Male	USA	2912	42	[(1, .12), (3, .15), (30, 1.41), ...]
2011-01-01T02:00:00Z	ultratrimfast.com	google.com	Male	UK	1953	17	[(2, .03), (1, .62), (20, .93), ...]
2011-01-01T02:00:00Z	bieberfever.com	google.com	Male	UK	3194	170	[(1, .05), (94, .84), (1, 1.14), ...]

图 7-2 ApproxHistogram 功能所需的数据格式

比如第一行中 AH_Prices 列中，[(1, .16), (48, .62), (83, .71), ...]，表示值为 0.16 的有 1 个；重心为 0.62 的有 48 个；重心为 0.71 的有 83 个等等，当有新的数据插入时，这些二元组也会重新计算值。

在计算直方图时，可以利用这些二元组进行近似计算。具体过程是，根据这些二元组构造每一区段的值的预估个数，就可以得到直方图和分位数，如图 7-3 所示。

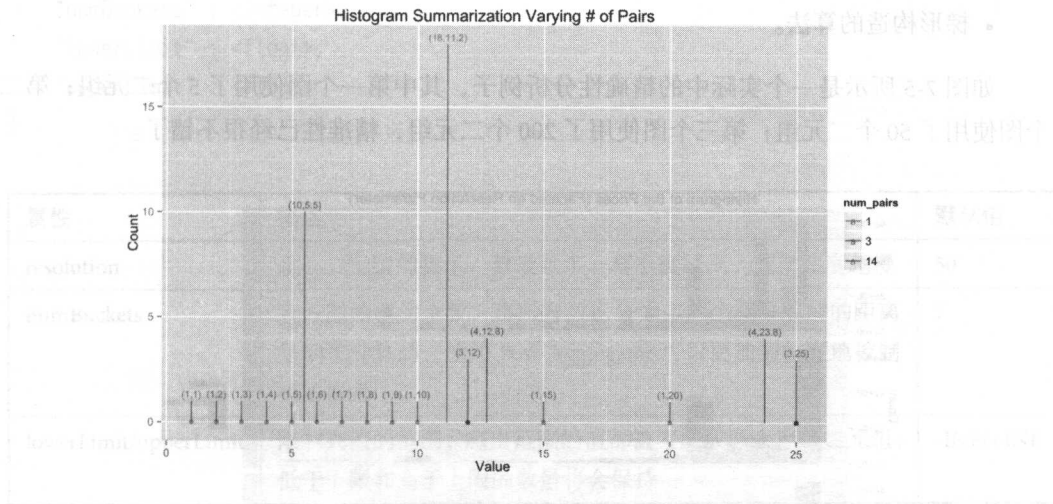


图 7-3 直方图和分位数样例

预估的梯形如图 7-4 所示，其中包括预估的一些值。

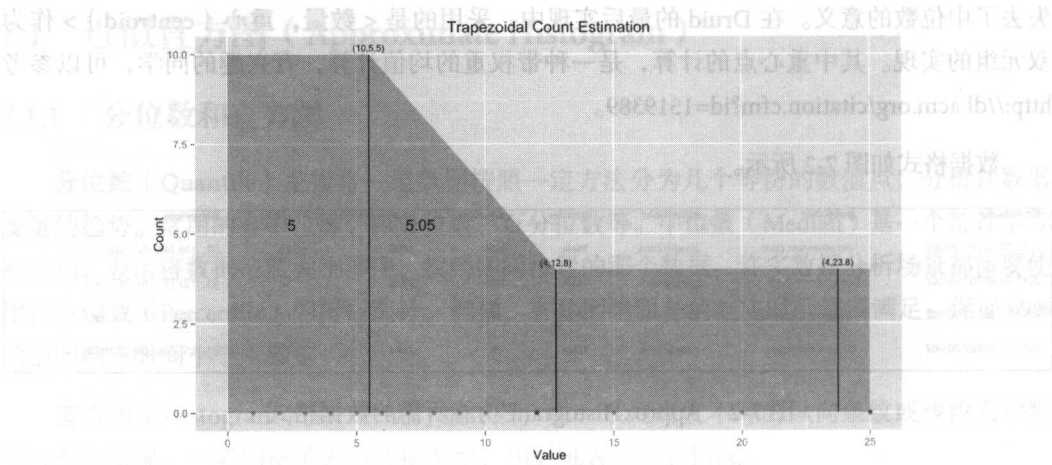


图 7-4 预估的梯形图

如下几个因素决定了预估的精准性。

- 二元组的最大数量。
- 数据分布的规律性。
- 梯形构造的算法。

如图 7-5 所示是一个实际中的精确性分析例子，其中第一个图使用了 5 个二元组；第二个图使用了 50 个二元组；第三个图使用了 200 个二元组，精准性已经很不错了。

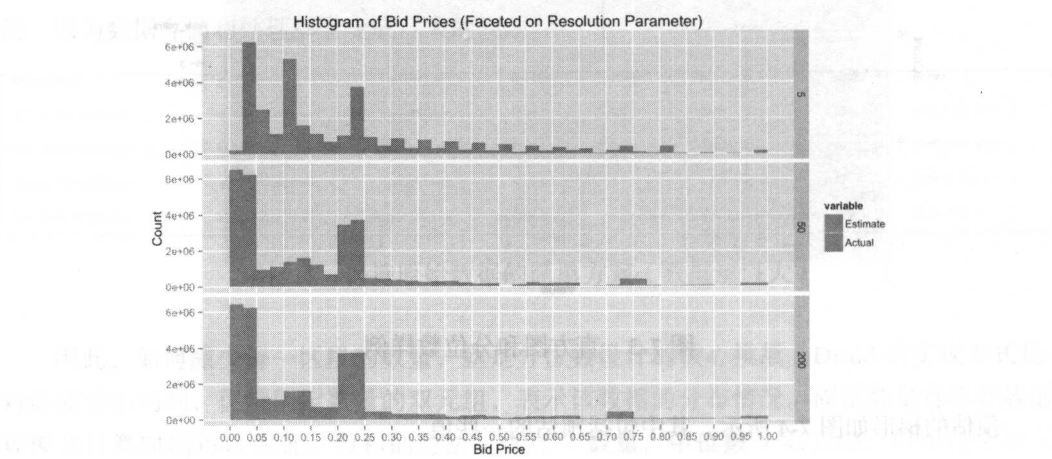


图 7-5 使用了不同二元组的精确性分析例子

7.1.3 如何使用

下面介绍直方图和分位数的具体使用。

1. 索引阶段

为了使用这个功能，在索引时需要打开“ApproxHistogram”或者“ApproxHistogramFold”聚合器。这种摄入聚合器只能用于数字的列值。这两种聚合器的区别如下：

- ApproxHistogram——如果数据行缺少此列值，这个值被认为是 0。
- ApproxHistogramFold——如果数据行缺少此列值，这个值将不加入计算（Ignored）。

2. 查询

在查询输入中必须指定“ApproxHistogramFold”聚合器，例子如下：

```
{
  "type" : "approxHistogram or approxHistogramFold (at ingestion time / at query time)",
  "name" : <output_name>,
  "fieldName" : <metric_name>,
  "resolution" : <integer>,
  "numBuckets" : <integer>,
  "lowerLimit" : <float>,
  "upperLimit" : <float>
}
```

属性	描述	默认值
resolution	重心二元组的数量；数量越多，精准度越高，查询速度越慢	50
numBuckets	直方图的桶的个数，将所有数据分为多少个部分。桶的距离是动态计算的，使用 Post-Aggregator 可以更加细粒度地控制桶的格式	7
lowerLimit/upperLimit	限制列值的范围，超出范围的值都被考虑成两个重心二元组，低于下限和高于上限的数量将会保持	-INF/+INF

3. 近似直方图的 Post-Aggregator

Post-Aggregator 用于将近似直方图的概要数据转化成分段式直方图表示，并且计算不同的分布式指标，例如百分位数、最大值、最小值等。

(1) 相同桶 (equalBuckets) Post-Aggregator

这个 Post-Aggregator 的功能是可以指定相同大小的桶，只需要指定桶的数量就可以，桶内部具体大小取决于底层的数据内容。

```
{
  "type": "equalBuckets",
  "name": "<output_name>",
  "fieldName": "<aggregator_name>",
  "numBuckets": <count>
}
```

(2) 自定义桶 (buckets) Post-Aggregator

这个 Post-Aggregator 的功能是可以指定桶的开始大小、桶的步长大小和桶的个数。

```
{
  "type": "buckets",
  "name": "<output_name>",
  "fieldName": "<aggregator_name>",
  "bucketSize": <bucket_size>,
  "offset": <offset>
}
```

(3) 最小 Post-Aggregator

返回近似直方图最小值。

<pre>{ "type": "min", "name": "<output_name>", "fieldName": "<aggregator_name>" }</pre>		<p>返回近似直方图最小值。</p>
<p>(4) 最大 Post-Aggregator</p>		<p>返回近似直方图最大值。</p>

```
{
  "type": "max",
  "name": "<output_name>",
  "fieldName": "<aggregator_name>"
}
```

(5) 单分位数 Post-Aggregator

指定分位数的结果。

```
{
  "type" : "quantile",
  "name" : <output_name>,
  "fieldName" : <aggregator_name>,
  "probability" : <quantile>
}
```

(6) 多分位数 Post-Aggregator

多分位数用于计算多个分位数的 Post-Aggregator，其用法和单分位数类似，只是支持多个输入。

```
{
  "type" : "quantiles",
  "name" : <output_name>,
  "fieldName" : <aggregator_name>,
  "probabilities" : [ <quantile>, <quantile>, ... ]
}
```

7.1.4 近似直方图小结

由于 Druid 的数据在存储时进行了预聚合操作，因此对于 Druid 来说，一些常见的数据库操作的分析实现起来就非常难了。为了实现这些功能，Druid 不得不进行准确性和性能的权衡。扩展的直方图还是可以方便地实现直方图和分位数的分析目标的。

7.2 数据 Sketch

7.2.1 DataSketch Aggregator

Druid DataSketch 是基于 Yahoo 开源的 Sketch 包 (<http://datasketches.github.io/>) 实现的数
据近似计算功能。Druid DataSketch 能够实现快速的纬度基数运算，后面使用的 ThetaSketch 算
法会对 Segment 索引列的每一段构造一个数据索引结构（元数据 + Key-Value 对）。ThetaSketch
使用概率论方法对基数进行统计分析。在前面的章节中，我们谈到过 HyperUnique Aggregator
功能，其实现是使用 HyperLogLog 算法。下面做一个简单的比较。

两个功能的相同之处：都能实现基本的近似基数运算。

不同之处：

- DataSketch 的基数运算比 HyperUnique 丰富：支持集合的交、并、补运算。
- DataSketch 使用扩展插件实现，HyperUnique 为内置功能。
- DataSketch 的精度高于 HyperLoglog，并且可以灵活进行参数调整。

下面举一个例子，查询既浏览了 A 商品又浏览了 B 商品的用户数是多少。

要使用 DataSketch Aggregator，必须在 Druid 启动时把 Druid DataSketch Aggregator 插件加上：

```
druid.extensions.loadList=["druid-datasketches"]
```

在摄取时，需要指定 Sketch Aggregator：

```
{
  "type": "thetaSketch",
  "name": <output_name>,
  "fieldName": <metric_name>,
  "isInputThetaSketch": false,
  "size": 16384
}
```

其中，size 和结果的准确度有关，值越大，越准确。但是 size 值越大，就需要越多的存储数据，查询数据也会变慢。这需要使用者找到平衡。

我们仍以用户行为数据摄取为案例，在摄取阶段，配置如下：

```
{
  "type": "index",
  "spec": {
    "dataSchema": {
      "dataSource": "dianshang_order",
      "parser": {
        "type": "string",
        "parseSpec": {
          "format": "json",
          "timestampSpec": {
            "column": "timestamp",
            "format": "auto"
          }
        }
      }
    }
  }
}
```

```

    },
    "dimensionsSpec":{
      "dimensions":[
        "event_name",
        "age",
        "city",
        "commodity",
        "category"
      ],
      "dimensionExclusions":[
      ],
      "spatialDimensions":[
      ]
    }
  },
  "metricsSpec":[
    {
      "type":"longSum",
      "name":"count",
      "fieldName":"count"
    },
    {
      "type":"thetaSketch",
      "name":"theta_user_id",
      "fieldName":"user_id"
    }
  ],
  "granularitySpec":{
    "type":"uniform",
    "segmentGranularity":"HOUR",
    "queryGranularity":"MINUTE",
    "intervals":[
      "2016-08-27/2016-08-28"
    ]
  }
}

```

```

    },
    "ioConfig": {
        "type": "index",
        "firehose": {
            "type": "local",
            "filter": "*.json",
            "baseDir": "/rc/data/druid/tmp/test-data"
        }
    },
    "tuningConfig": {
        "type": "index",
        "maxRowsInMemory": 500000,
        "intermediatePersistPeriod": "PT10m",
        "windowPeriod": "PT10m",
        "basePersistDirectory": "/rc/data/druid/realtime/basePersist",
        "rejectionPolicy": {
            "type": "none"
        }
    }
}

```

样例数据如下：

```

{"timestamp": "2016-08-27T08:50:00.563Z", "event_name": "browse_commodity", "user_id": 1, "age": "90+", "city": "Beijing", "commodity": "A", "category": "3c", "count": 2}
{"timestamp": "2016-08-27T08:50:00.563Z", "event_name": "browse_commodity", "user_id": 2, "age": "90+", "city": "Beijing", "commodity": "A", "category": "3c", "count": 1}
{"timestamp": "2016-08-27T08:50:00.563Z", "event_name": "browse_commodity", "user_id": 1, "age": "90+", "city": "Beijing", "commodity": "A", "category": "3c", "count": 1}
{"timestamp": "2016-08-27T08:50:00.563Z", "event_name": "browse_commodity", "user_id": 3, "age": "90+", "city": "Beijing", "commodity": "A", "category": "3c", "count": 1}
...

```

我们可以通过如下查询，查询 UV：

```

{
    "queryType": "timeseries",
    "dataSource": "dianshang_order",
    "granularity": "day",

```

```

"aggregations": [
  { "type": "thetaSketch", "name": "theta_user_id", "fieldName": "theta_user_id" }
],
"intervals": [ "2016-08-27/2016-08-28" ]
}

```

得到结果如下:

```

[[
  {
    "timestamp" : "2016-08-27T00:00:00.000Z",
    "result" : {
      "theta_user_id" : 7.0
    }
  }
]]

```

7.2.2 DataSketch Post-Aggregator

如前所述, DataSketch 还能查询既浏览了 A 商品又浏览了 B 商品的用户数。这就需要使
用 Post-Aggregator。

1. Sketch Estimator

Sketch Estimator 的使用如下:

```

{
  "type" : "thetaSketchEstimate",
  "name": <output name>,
  "field" : <post aggregator of type fieldAccess that refers to a thetaSketch
            aggregator or that of type thetaSketchSetOp>
}

```

Sketch Estimator 用于计算 Sketch 的预估值。

2. Sketch Operation

Sketch Operation 的使用如下:

```

{
  "type" : "thetaSketchSetOp",
  "name": <输出名字>,
  "func": <UNION|INTERSECT|NOT>, //并、交、补

```

```

"fields" : <...>, //所需要操作的列名数组
"size": <16384> //ThetaSketch的数据包大小, 取值为2的指数整数, 默认值为16384
}

```

Sketch Operation 用于 Sketch 的集合运算。

那么查询既浏览了 A 商品又浏览了 B 商品的用户数如下:

```

{
  "aggregations": [
    {
      "aggregator": {
        "fieldName": "theta_user_id",
        "name": "A_theta_user_id",
        "type": "thetaSketch"
      },
      "filter": {
        "dimension": "commodity",
        "type": "selector",
        "value": "A"
      },
      "type": "filtered"
    },
    {
      "aggregator": {
        "fieldName": "theta_user_id",
        "name": "B_theta_user_id",
        "type": "thetaSketch"
      },
      "filter": {
        "dimension": "commodity",
        "type": "selector",
        "value": "B"
      },
      "type": "filtered"
    }
  ],
  "dataSource": "dianshang_order",
  "dimensions": [],

```

```

"filter": {
  "fields": [
    {
      "dimension": "commodity",
      "type": "selector",
      "value": "A"
    },
    {
      "dimension": "commodity",
      "type": "selector",
      "value": "B"
    }
  ],
  "type": "or"
},
"granularity": "day",
"intervals": [
  "2016-08-27/2016-08-28"
],
"postAggregations": [
  {
    "field": {
      "fields": [
        {
          "fieldName": "A_theta_user_id",
          "type": "fieldAccess"
        },
        {
          "fieldName": "B_theta_user_id",
          "type": "fieldAccess"
        }
      ],
      "func": "INTERSECT",
      "name": "final_unique_users_sketch",
      "type": "thetaSketchSetOp"
    },
    "name": "final_unique_users",
    "type": "thetaSketchEstimate"
  }
]

```



```

    }
  ],
  "queryType": "groupBy"
}

```

查询结果如下:

```

[[{
  "event": {
    "A_theta_user_id": 3.0,
    "B_theta_user_id": 6.0,
    "final_unique_users": 2.0
  },
  "timestamp": "2016-08-27T00:00:00.000Z",
  "version": "v1"
}]

```

7.3 地理查询 (Geographic Query)

Druid 支持空间索引列, 数据列基于空间坐标或者区域范围。与其他几个扩展不一样, Druid 的空间索引为系统内置功能, 支持一些简单的空间索引和查询。

7.3.1 基本原理

Druid 的空间索引设计使用的是常规的 R-Tree 数据结构, 支持多维空间坐标, 从简单的二维平面、三维空间数到更多的纬度。R-Tree 的基本原理就是将空间中的临近点, 通过最小的长方形/长方体等包围起来, 并且建立树状索引, 在查询时利用二叉树的查询方法, 快速过滤节点。

Druid 的实现方式是直接使用 MetaMarkets 公司的数据结构库, 该库的实现可以从 <https://github.com/metamx/bytebuffer-collections> 地址获得。其关键实现为 `ImmutableRTree.java`。这个数据结构代表一个压缩成 Bitmap 的 R-Tree 结果, 用于 Segment 中列的序列化和反序列化。

空间索引的索引文件为独立文件, 文件名为 `spatial.drd`。

7.3.2 空间索引 (Spatial Indexing)

在数据定义中，Druid 提供了空间纬度的定义。下面是一个例子，描述如何定义空间索引。

```
"dataSpec" : {
  "format": "JSON",
  "dimensions": <some_dims>,
  "spatialDimensions": [
    {
      "dimName": "coordinates",
      "dims": ["lat", "long"]
    },
    ...
  ]
}
```

属性	描述	是否必需
dimName	空间维度的名字。空间维度可以从其他维度构造，或者从已经存在的维度构造。如果一个空间纬度已经存在，那么它必须是坐标值的数组	是
dims	包含空间维度的名字列表	否

7.3.3 空间过滤 (Spatial Filter)

空间过滤的语法如下：

```
"filter" : {
  "type": "spatial",
  "dimension": "spatialDim",
  "bound": {
    "type": "rectangular",
    "minCoords": [10.0, 20.0],
    "maxCoords": [30.0, 40.0]
  }
}
```

7.3.4 边界条件 (Boundary Condition)

目前支持两种基本边界条件：一是长方形；二是半径。

1. 长方形 (Rectangular)

属性	描述	是否必需
minCoords	最小坐标轴列表 [x, y, z, ...]	是
maxCoords	最大坐标轴列表 [x, y, z, ...]	是

2. 半径 (Radius)

属性	描述	是否必需
coords	原点的坐标 [x, y, z, ...]	是
radius	浮点表示的半径值	是

7.3.5 地理查询小结

目前，Druid 虽然支持空间索引，但是坐标系还是基于笛卡儿坐标，而不是真正的地理坐标，例如经度、纬度等。支持的查询过滤条件也比较少（长方形或半径），地理索引的性能也慢于很多专业的空间数据库，该功能的应用范围还不广，多用作分析数据库的一个补充，例如 IP 库的一些附属索引信息。

7.4 Router

当集群规模达到 TB 级别时，就可以考虑使用 Router（路由器）。Router 用于把查询路由到不同的 Broker。它用来实现 Broker 层面的查询隔离，例如把热数据查询路由到指定的 Broker 集合，把不重要的查询路由到剩下的 Broker 集合。这种隔离方式在某些场景下非常有用，例如某些不重要的查询耗时过长，会影响其他重要查询的情况。

7.4.1 Router 概览

Router 是 Broker 之前的一层代理，它接收来自于客户端的请求，根据路由规则获取指定的 Broker，采用 Jetty 提供的异步、非阻塞的 HttpClient 发送请求，然后把返回的结果集发

送到客户端。接下来我们就通过经典的冷热数据分层的例子来讲解 Router 的架构及其使用。在生产环境中有两个 tier（层），即“hot”和“_default_tier”，“hot”层加载最近一个月的热数据，“_default_tier”层加载所有的数据。希望将“hot”层上的查询路由到“broker:hot”的 Broker 集合，将“_default_tier”层上的查询路由到“broker:cold”的 Broker 集合。整体架构示意图如图 7-6 所示。

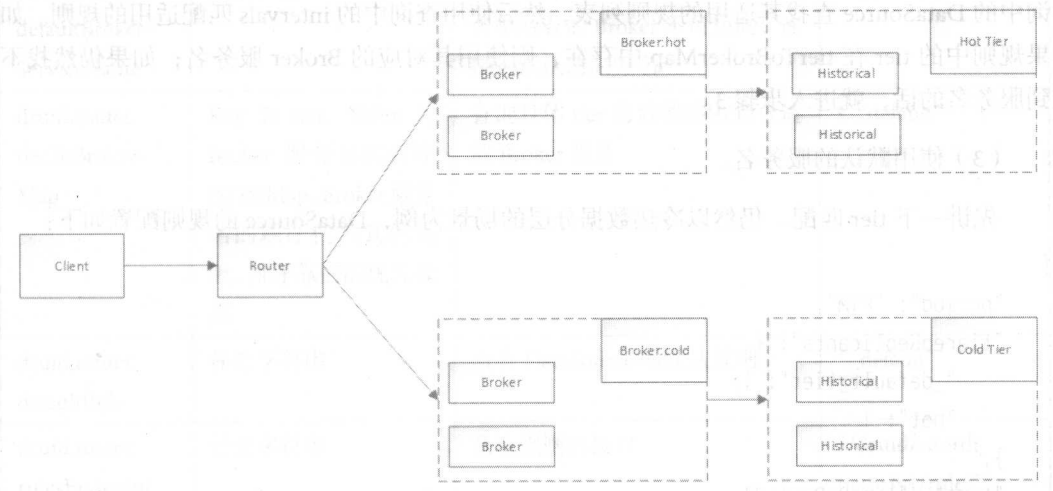


图 7-6 整体架构示意图

在 Router 中利用 CuratorDiscovery 服务发现机制，根据服务名对 Broker 进行分组，将具有相同服务名的 Broker 分到同一组。在 Broker 配置服务名的方式如下：

```
druid.service=broker:hot
```

当前服务名为“broker:hot”的 Broker 启动时会在 `${druid.discovery.curator.path}/broker/hot` 目录下创建临时节点存放自己的 Host 和 Port 等元数据信息。使用临时节点的好处是 Curator 会自动探测新加入的节点，以及剔除失败的节点，只有当同一服务名有多个 Broker 节点时，才用 RoundRobin 方式选取。

Router 中两个重要的配置项如下：

- `druid.router.tierToBrokerMap`，有序的 Map，Map 中元素的顺序代表优先级，排序越靠前，优先级越高。键是 tier，是我们在历史节点中配置的层，值是 Broker 的服务名。
- `druid.router.defaultBrokerServiceName`，默认的服务名，当所有的路由规则都满足不了，或者选取的服务名中所有 Broker 都不可用时，使用默认的服务名。

7.4.2 路由规则

(1) 首先使用 `$(druid.router.strategies)` 中配置的路由策略列表逐个去匹配，如果能匹配上，就跳过以后的策略；如果匹配不上，则进入步骤 2。

(2) 使用 `tier` 匹配，Router 会定期调用 Coordinator 的接口获取所有的规则，首先根据查询中的 `DataSource` 查找其适用的规则列表，然后使用查询中的 `intervals` 匹配适用的规则。如果规则中的 `tier` 在 `tierToBrokerMap` 中存在，则使用其对应的 Broker 服务名；如果仍然找不到服务名的话，就进入步骤 3。

(3) 使用默认的服务名。

先讲一下 `tier` 匹配。仍然以冷热数据分层的场景为例，`DataSource` 的规则配置如下：

```
{
  "period": "P1M",
  "tieredReplicants": {
    "_default_tier": 1,
    "hot": 1
  },
  "type": "loadByPeriod"
}
```

将最近一个月的数据，分别加载到“hot”和“_default_tier”两个层中。`tierToBrokerMap` 的配置如下：

```
{ "hot": "broker:hot", "_default_tier": "broker:cold" }
```

规则中的 `Period` 为“P1M”，表示规则适用的 `intervals` 为最近一个月，和查询的 `intervals` 重合，说明查询适用该规则。接下来按照顺序遍历 `tierToBrokerMap`，检查 `tier` 是否出现在规则的 `tieredReplicants` 中，如满足则跳出循环。

```
for (Map.Entry<String, String> entry : tierConfig.getTierToBrokerMap().entrySet()) {
  if (baseRule.getTieredReplicants().containsKey(entry.getKey())) {
    brokerServiceName = entry.getValue();
    break;
  }
}
```

`tierToBrokerMap` 是有序的，元素的顺序代表了优先级，`tier` 匹配过程就是找到最高优先级（最先出现）的 Broker。举例来说，按照上述代码执行以后，获得的 `tier` 就是“broker:hot”。

7.4.3 配置

接下来，对配置中重要的属性进行讲解。

属性	可能的值	解释	默认值
druid.router.defaultBroker-ServiceName	任意字符串	路由规则不满足或者选取的服务名中所有的 Broker 不可用时，连接到默认的服务名	druid/broker
druid.router.tierToBroker-Map	Key 为 tier, Value 为 Broker 服务名的有序 JSON Map, Broker 服务名的顺序代表其优先级, 排序靠前的优先级高	查询具体 tier 的数据路由到合适的 Broker 服务	{"_default": ""}
druid.router.defaultRule	任意字符串	所有 DataSource 的默认规则	"_default"
druid.router.rulesEndpoint	任意字符串	获取规则的接口	"/druid/coordinator/v1/rules"
druid.router.coordinator-ServiceName	任意字符串	Coordinator 的服务名, 用于获取 Coordinator 节点	
druid.router.pollPeriod	Any ISO 8601 duration	轮询周期, 用于发现新的规则	PT1M
druid.router.strategies	有序的 JSON 数组	自定义的路由策略列表	[{"type": "time-Boundary"}, {"type": "priority"}]

7.4.4 路由策略

Router 需要配置一系列的路由策略, 用来选取具体的 Broker 服务名去执行查询。路由策略的顺序很重要, 因为一旦满足策略条件, 选取出 Broker 服务名后, 就不再执行后面的策略了。

1. timeBoundary

```
{
  "type": "timeBoundary"
}
```

添加该策略，意味着 `timeBoundary` 类型的查询总是路由到最高优化级的 Broker 服务名。

2. priority

```
{
  "type": "priority",
  "minPriority": 0,
  "maxPriority": 1
}
```

如果查询的优先级小于 `minPriority`，则会被路由到最低优先级的 Broker 服务名。默认地，将 `minPriority` 设置为 0，`maxPriority` 为 1，如果查询的优先级为 0，查询的默认优先级是 0，则会跳过该策略。

3. JavaScript

通过 JavaScript 函数的方式可以实现任意的路由规则，该函数需要传入 `config` 和 `query` 两个参数，返回 `tier` 所需要的 Broker 服务，如果没有则返回 Null，表示使用默认的 Broker 服务。

如下示例是如果查询中含有三个以上 Aggregator，则会被路由到最低优先级的服务名。

```
{
  "type": "javascript",
  "function": "function (config, query) {
    //为了便于演示，JavaScript代码添加换行和缩进，使用时需要符合JSON规范
    if (query.getAggregatorSpecs() && query.getAggregatorSpecs().size() >= 3) {
      var size = config.getTierToBrokerMap().values().size();
      if (size > 0) {
        //获取tierToBrokerMap的最后一个元素，排序越靠后，优先级越低
        return config.getTierToBrokerMap().values().toArray()[size-1]
      } else {
        return config.getDefaultBrokerServiceName()
      }
    } else {
      return null
    }
  }"
}
```

通过对上述路由策略的学习,知道其实现机制是根据查询的优先级、类型或者其他特性,例如 JavaScript 例子中的 Aggregator 数量,来选择合适优先级的 Broker 服务名, Broker 服务名的优先级是由其在 tierToBrokerMap 中的排序来决定的,排序靠前则说明优先级高。回到冷热数据分层的例子中, tierToBrokerMap 配置是 {"hot":"druid:broker-hot", "_default_tier":"druid:broker-cold"}, 我们想让时间跨度为最近一个月的查询落在 “druid:broker-hot”, 可以使用默认配置的 “priority” 路由策略, 同时将查询的优先级设置为 2, $2 > \text{maxPriority}(1)$, 则会使用最高优先级的服务名 “druid:broker-hot”。由此可见, 使用默认的路由策略, 并且不调整查询的优先级, 除了 timeBoundary 查询以外, 只能通过 tier 匹配的方式路由。

Router 相当于根据策略规则把查询路由到 Broker 的反向代理, 加入这一层以后势必会对查询性能造成影响。所以就像开始提到的那样, 当数量没有达到 TB 级别, 或者对隔离 Broker 层面的查询没有迫切需求时, 不推荐使用 Router。

7.5 Kafka 索引服务

7.5.1 设计背景

Kafka 索引服务为增强数据实时摄入而生。其核心特性如下。

- 保障数据摄入的 Exactly Once (有且只有一次)。实现 Exactly Once, 需要保障既不能重复摄入, 同时也不能丢弃任何一条数据。
- 不再受 windowPeriod 的约束, 可以摄入任意时间戳的数据, 而不仅仅是当前的数据。
- 操作的易用性, 自适应性强, 可以根据 Kafka 分区的增加或者减少调整任务的数量。

实现 Exactly Once 语义, 必须要保障不重复摄入数据, 以及不丢弃数据。

影响数据丢弃的主要因素是时间窗口 (windowPeriod), 时间窗口的设定是为了解决数据的延迟以及乱序问题, 它允许数据存在时间窗口内的延迟, 超出以后被丢弃。

windowPeriod 会影响 Segment 移交 (Hand off) 的时间点, 达到 Segment 设定的时间粒度以后延迟时间窗口的时间再进行移交, 这样可以保障延迟的数据依然能被添加到 Segment 中。分布式日志收集以及实时计算过程中都会导致延迟, 而且延迟是不可预测的, 所以很难找到一个合适的 windowPeriod 来保障不丢弃数据。你可能想, 把时间窗口调整为足够大不就解决问题了吗? 时间窗口调大的负面影响是延迟移交, 它会影响索引服务的任务的完成退出以及查询性能, 所以需要在延迟移交和丢弃数据之间进行权衡, 选择合适的时间窗口。

影响数据仅摄入一次、不重复摄入的主要因素是 Kafka 的 Offset（偏移）管理。在最初的 KafkaFireChief 实现中采用 High Level（高层）的 Consumer（消费者），它会帮助完成 Broker 的 Leader 选择、Offset 的维护、管理分区和消费者之间的均衡以及重平衡等功能。采用 Group（组）的方式，同一 Group 内的消息只能被一个消费者消费一次。但它也会带来一些问题。

- 同一 Group 内的消息只能被消费一次，所以很难实现数据摄入阶段的多副本来保障高可用性和查询一致性。
- High Level 的 Consumer 采用 Zookeeper 保存 Offset，内存增量索引持久化和 Offset 的提供是分开进行的，不能在同一事务中处理。假设持久化成功但还没有来得及提交 Offset，节点失败，然后重启，加载持久化的索引以后，继续从上一次提交的 Offset 读取导致数据重复摄入。虽然这种情况发生的概率很低，但还是有可能发生的。

7.5.2 实现

Kafka 索引服务为了实现 Exactly Once 语义，去掉了 windowPeriod 以及采用 Kafka 的底层 API 实现对 Offset 的事务管理。Kafka 索引服务采用 Supervisor（监督者）的方式运行在 Overlord 上。我们来看一下其实现的关键类。

KafkaSupervisor 类似于一个大管家，负责 Kafka 索引任务的创建以及管理其整个生命周期，它会监管 Kafka 索引任务的状态来完成协调移交、管理失败，同时保障添加/删除 Kafka 分区以后的可扩展性，以及任务多副本执行。

KafkaPartitions 用来记录 Kafka 的 Topic（主题）以及 Partition->Offset 的映射关系的数据结构。

KafkaIndexTask（Kafka 索引任务）从 KafkaIOConfig 的 startPartition 中的 Offset 处开始读取数据，一直到 endPartition 的结束 Offset 处以后结束读取，发布移交 Segment。在执行过程中，startPartition 中的 Offset 不会改变，endPartition 的 Offset 初始设置为 Long.MAX_VALUE，KafkaSupervisor 通过修改 endOffset 的值来结束任务的执行。运行中的任务一般有两种状态：读取和发布。任务会保持读取状态，直到达到 taskDuration 以后进入发布状态。接下来会保持发布状态，直到生成 Segment，并推送到深度存储中，以及等待历史节点加载以后（或者达到 completionTimeout 的时间）。

TaskGroup 是 KafkaSupervisor 管理 Kafka 的分区、分区的 Offset 以及 Kafka 索引任务的重要数据结构，用来保障任务多副本执行。TaskGroup 中的所有任务总是做相同的事情，读取分配给 TaskGroup 的所有 Kafka 分区的数据，而且都是从相同的起始 Offset 处读取。通过修改 replicas 的值来调整任务副本的数量。

Appenderator 用来索引数据,采用类似于 LSM-Tree 的架构。它由内存增量索引和持久化索引组成,并负责这两部分的查询。这个模块负责索引和查询数据,并将 Segment 推送到深度存储中,它并不负责 Segment 的分配,而只是将 Segment 信息发布到元数据存储中。

FiniteAppenderatorDriver 驱动 Appenderator 完成有限流式数据的索引,它不能处理无边界流式数据,在索引结束以后执行移交操作。在这个类中完成 Appenderator 不能做的那些事情,包括使用 SegmentAllocator 将数据分配到指定的 Segment,以及监控移交等工作。

SegmentAllocator 根据给定的时间戳,分配一个 Segment。

KafkaSupervisor 负责管理单个数据源的 Kafka 索引任务。它会定期执行如下过程。

从 Kafka 中更新 Partition 信息,用于发现新的 Partition,并为其分配 TaskGroup。分配规则如下:

```
taskGroupId = partition % ioConfig.getTaskCount();
```

检查是否达到任务的持续时间,任务的持续时间由参数“taskDuration”来决定,默认为 1 小时。如果达到了,则发送信号提示其停止读取数据,进入 Segment 发布阶段。其执行流程如图 7-7 所示。

KafkaSupervisor 有三个重要的数据结构。

- taskGroups, 保存读取状态的 TaskGroup 的 Map 结构。Key 为 groupId, Value 为相应的 TaskGroup。
- pendingCompletionTaskGroups, 使用信号通知 Task Group 结束读取,进入 Segment 发布状态以后,将该 Task Group 从 taskGroups 移除放入 pendingCompletionTaskGroups 中。
- partitionGroups, 保存 taskGroupID 和 Kafka 分区起始偏移的 Map 结构。其结构为 Map<{group ID}, Map<{partition ID}, {startingOffset}>>。

下面重点介绍信号通知的实现机制。

- 暂停 Task Group 中的所有任务, KafkaIndexTask 提供了“/pause”接口,调用该接口可以暂停从 Kafka 中读取数据,并返回该任务的 currentOffsets, currentOffsets 包含了所有 Partition->Offset 的映射关系。比较 TaskGroup 中每个任务返回的 currentOffsets,找出每个 Partition 对应的最高 Offset,构建成 Map 结构。这里把它叫作 endOffsets。
- 调用 KafkaIndexTask 的“/offsets/end”接口,设置 endOffsets 为上一步得到的值,并恢复其执行。当任务的读取迭代过程发现达到 endOffsets 时,会跳出迭代进入 Segment 发布状态。

- 将 endOffsets 作为下一个 TaskGroup 的起始 Offset 放入 partitionGroups 中，用于创建新的 Task Group 继续从前一个任务的结束 Offset 处读取数据。这样设计的好处是不必等任务完成以后就可以执行下一个任务。

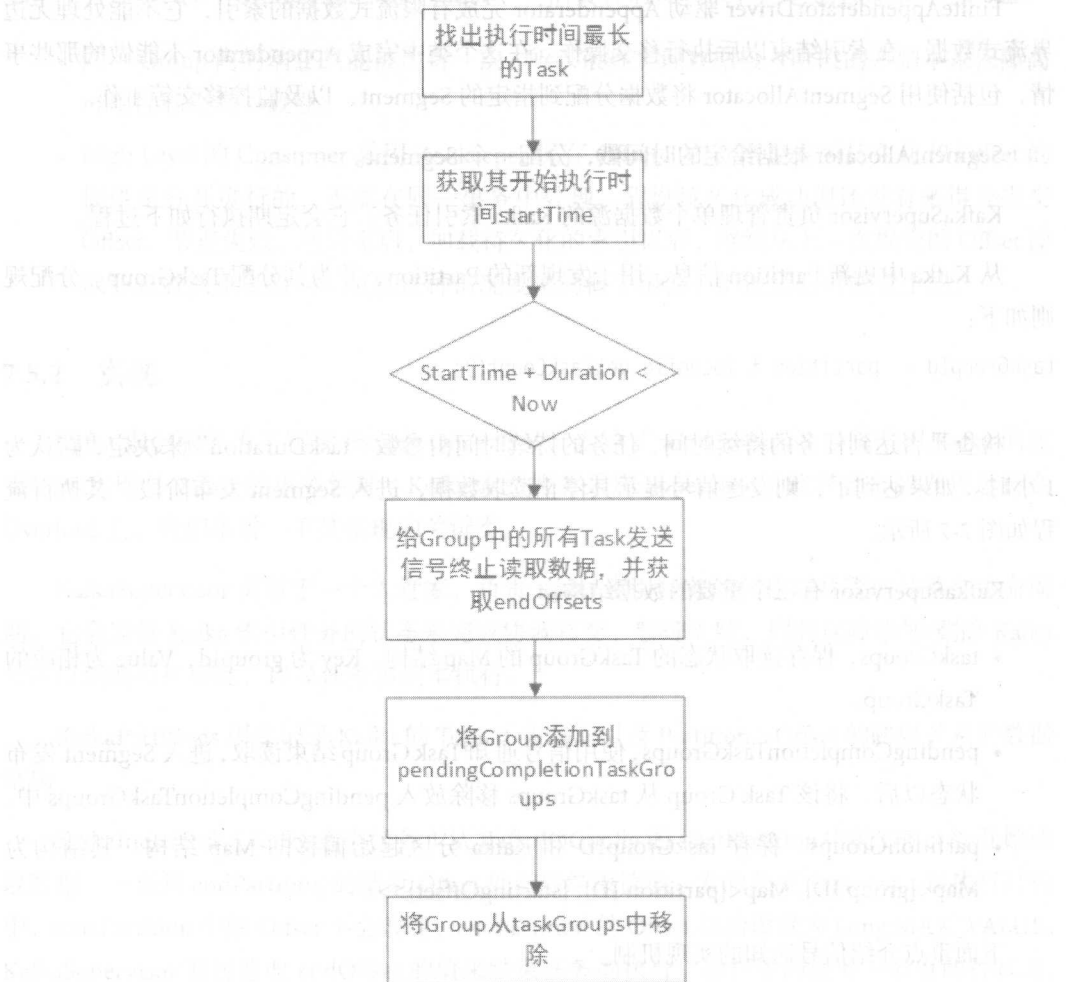


图 7-7 终止 Task Group 的流程图

创建 TaskGroup 及其任务副本，Supervisor 会定期检测从 partitionGroups 的 keySet 中遍历，查找有哪些 groupId 还没有创建 TaskGroup，执行过程请参照如下代码。

```
for (Integer groupId : partitionGroups.keySet()) {  
    if (!taskGroups.containsKey(groupId)) {  
        taskGroups.put(groupId, new TaskGroup(generateStartingOffsetsForPartitionGroup(  

```

```

        groupId), minimumMessageTime));
    }
}

```

从图 7-7 得知, 当通知任务结束读取以后, 会将 Task Group 从 taskGroups 中移除, 同时把 partitionGroups 中的起始 Offset 设置为上个任务的 endOffsets。然后执行如上过程, 创建新的 TaskGroup 延续读取数据。随后根据 replicas 的配置, 创建任务副本。

Kafka 索引任务利用 Kafka 的底层 Consumer 接口, 从传入的 startOffsets Map{partition, start-Offset} 处读取数据, 使用 FiniteAppenderatorDriver 构建 Segment。FiniteAppenderatorDriver 摒弃了采用时间窗口处理无边界流式数据的方式, 而是采用有限流式数据的处理方式, 读取固定时间段的数据。当从 Kafka 中读取一条数据以后, 不再判断是否超出“时间窗口”来决定留下还是丢失, 而是根据数据的时间戳使用 SegmentAllocator 分配到合适的 Segment。我们来看一个例子, 假设 segmentGranularity 为 HOUR, 如下数据为实时数据, 在 2 点钟开始的边界时刻过来 1 点钟的数据, 这是典型的延迟和乱序问题。如果使用时间窗口的方式, 1 点钟的数据会被丢弃, 后者则会分配到两个 Segment 而不会丢弃数据。

timestamp	publisher	advertiser	gender	country	click	price
2016-01-01T02:01:35Z	bieberfever.com	google.com	Male	USA	0	0.65
2016-01-01T02:03:63Z	bieberfever.com	google.com	Male	USA	0	0.62
2016-01-01T02:04:51Z	bieberfever.com	google.com	Male	USA	1	0.45
2016-01-01T01:52:00Z	ultratrifast.com	google.com	Female	UK	0	0.87
2016-01-01T01:53:00Z	ultratrifast.com	google.com	Female	UK	0	0.99
2016-01-01T01:51:00Z	ultratrifast.com	google.com	Female	UK	1	1.53

Segment 的 intervals 分别为 2016-01-01T01:00:00Z_2016-01-01T02:00:00Z 和 2016-01-01T02:00:00Z_2016-01-01T03:00:00Z。通常 Segment 标识符的命名规则为“interval_partitionNum”, 但如果 FiniteAppenderatorDriver 继续使用这种方式, 则可能会导致标识符冲突。依然是如上的数据, 1 点钟的数据已经在上一时刻创建 Segment, 标识符为“2016-01-01T01:00:00Z_2016-01-01T02:00:00Z_1”, 其中 1 为 partitionNum。在 2 点钟的某一时刻过来的 1 点钟的数据, 为其创建的 Segment 的标识符也可能为“2016-01-01T01:00:00Z_2016-01-01T02:00:00Z_1”, 标识符发生冲突。为了解决这一问题, 基于任务的 startPartitions 中的 Partition 和 startOffset 再加上其他参数生成 Hash 值, 截取前 16 位作为 Sequence, 添加到标识符中。不同 Task Group 的 Partition 和 startOffset 不同, 所以采用 Sequence 的方式能避免标识符冲突。

FiniteAppenderatorDriver 会带来副作用, 会产生一些碎片化的 Segment, 特别是日志跨时段乱序延迟严重的情况。

去掉时间窗口，解决了数据丢失的问题，实现 Exactly Once，还需要解决不重复摄入数据的问题。实现不重复摄入数据的关键因素是 Segment 信息发布到元数据存储中和 Offset 的提交需要在同一事务中处理。FiniteAppenderatorDriver 在索引的构建过程中不会修改 startOffset，只有在发布阶段，将 endOffset 写入元数据存储中，Segment 信息也写入同一个元数据存储中，元数据存储一般采用 MySQL 等关系型数据库，使用事务处理的方式解决重复摄入数据的问题。

7.5.3 如何使用

首先要将“kafka-indexing-service”的核心扩展添加到 Overlord（统治节点）和 Middle Manager（中间管理者）中。

```
druid.extensions.loadList=["other-module", "kafka-indexing-service"]
```

Kafka 索引服务使用 Kafka 0.9 版本的 Consumer API，由于在 0.9 版本中协议改变了，不兼容之前版本的 Kafka Broker，所以最好将 Kafka Broker 升级到 0.9 版本。

编写 Supervisor 规范，通过 HTTP POST 方式提交给 Overlord 的 URL: `http://<OVERLORD_IP>:<OVERLORD_PORT>/druid/indexer/v1/supervisor`。

如何编写 Supervisor 规范，我们来看一个简单的样例。

```
{
  "dataSchema": {
    "dataSource": "metrics-kafka",
    "granularitySpec": {
      "queryGranularity": "NONE",
      "segmentGranularity": "HOURL",
      "type": "uniform"
    },
    "metricsSpec": [
      {
        "name": "count",
        "type": "count"
      }
    ],
    "parser": {
      "parseSpec": {
        "dimensionsSpec": {
          "dimensionExclusions": [
```

```
        "timestamp",
        "value"
    ],
    "dimensions": [],
  },
  "format": "json",
  "timestampSpec": {
    "column": "timestamp",
    "format": "auto"
  }
},
"type": "string"
}
},
"ioConfig": {
  "consumerProperties": {
    "bootstrap.servers": "localhost:9092"
  },
  "replicas": 1,
  "taskCount": 1,
  "taskDuration": "PT1H",
  "topic": "metrics"
},
"tuningConfig": {
  "maxRowsPerSegment": 5000000,
  "type": "kafka"
},
"type": "kafka"
}
```

类似其他的摄入规范, Supervisor 规范由 type、dataSchema、tuningConfig 和 ioConfig 组成。

属性	解释	是否必需
type	Supervisor 的类型, 对于 Kafka 索引服务而言, 必须是 “kafka”	是
dataSchema	用于 KafkaIndexTask 摄入数据, 配置和其他摄入规范相同	是
tuningConfig	KafkaTuningConfig, 用于给 KafkaIndexTask 调优	否
ioConfig	KafkaSupervisorIOConfig, 用于配置 Supervisor	是

KafkaTuningConfig 详解如下：

属性	类型	解释	是否必需
task	String	索引任务的类型，必须是“kafka”	是
maxRowsIn-Memory	Integer	持久化之前内存增量索引中的最大条数。这是聚合以后的条数，不是摄入的原始数据集大小。调整该值，要确保有足够大的堆内存：(maxRows-InMemory × (2 + maxPendingPersists))	否（默认值：75000）
maxRowsPer-Segment	Integer	Segment 单个分片中的条数，注意是聚合以后的条数。超出了创建新的分片	否（默认值：5000000）
intermediate-PersistPeriod	ISO 8601 Period	中间持久化的发生频率	否（默认值：PT10M）
maxPending-Persists	Integer	当持久化的数量超出该值后，数据摄入会阻塞，直到当前执行的持久化完成	否（默认值为0，意味着所有的持久化都可以并发执行，而不需要排队等待）
reportParse-Exceptions	Boolean	在解析过程中发生异常后，如果该值为 true，则会抛出异常并挂起摄入；如果为 false，则会丢弃解析异常的数据继续执行	否（默认值：false）
handoffCondi-tionTimeout	Integer	等待 Segment 移交的超时时间，单位为毫秒，0 意味着永远等待	否（默认值：0）

KafkaSupervisorIOConfig 配置详解如下：

属性	类型	解释	是否必需
topic	String	从 Kafka 中读取数据的 Topic	否
consumerProperties	Map	传给 Kafka Consumer 的属性 Map。它至少包含 bootstrap.servers 属性，它是 Kafka Broker 的列表，格式为 BROKER_1:PORT_1,BROKER_2:PORT_2,...	是

续表

属性	类型	解释	是否必需
replicas	Integer	任务副本集合中的数量, 如果设置为 1, 则代表单个任务副本, 多个任务副本会被分配到不同的节点以保障高可用性	否 (默认值: 1)
taskCount	Integer	处于读取状态的任务总数的最大值, 不包括副本。这就意味着任务总数的最大值为 $\text{taskCount} \times \text{replicas}$, 甚至任务总数还会更大, 因为发布 Segment 时, 执行 Segment 发布的任务和继续读取数据的下一个任务同时存在。如果 $\text{taskCount} < \text{kafkaPartitionNum}$, 则会有单个任务处理多个 Partition; 反之, 任务的数量会小于 taskCount	否 (默认值: 1)
taskDuration	ISO 8601 Period	任务从开始执行到终止读取进入 Segment 发布状态的时间长度, Segment 推送到深度存储中以及被历史节点加载以后任务才算完成	否 (默认值: PT1H)
startDelay	ISO 8601 Period	Supervisor 管理任务之前的延迟	否 (默认值: PT5S)
period	ISO 8601 Period	Supervisor 定时执行管理逻辑的时间间隔, 需要注意的是, 它同时负责处理其他事件, 如任务的成功、失败以及检查是否达到持续时间, 所以这只是最大时间间隔, 如果处理上述事件则可能小于该值	否 (默认值: PT30S)
useEarliestOffset	Boolean	在 Supervisor 初始执行时, 假设元数据存储中还没有保存 Offset, 则从 Kafka 中获取 Offset, 如果该值为 true, 则使用 earliest 从最开始读取; 反之, 使用 largest, 从最新开始读取	否 (默认值: false)
completionTimeout	ISO 8601 Period	任务进入发布状态到发布成功, 然后任务退出的超时时间, 如果该值设为较小值, Segment 有可能永远不会发布	否 (默认值: PT30M)
lateMessage-RejectionPeriod	ISO 8601 Period	如果消息的时间戳早于 (任务的开始时间 - lateMessageRejectionPeriod), 则会被丢弃	否

7.6 Supervisor API

7.6.1 创建 Supervisor

POST /druid/indexer/v1/supervisor

Content-Type 设置为 application/json，把 Supervisor 规范放在 Request 请求体中。调用该接口时，如果相同数据源的 Supervisor 已经存在，则会导致：

- 正在运行的 Supervisor 会通知其管理的所有任务终止读取并执行 Segment 发布。
- 退出正在执行的 Supervisor。
- 使用 Request 请求体内的规范创建一个新的 Supervisor，它会接管处于发布状态的任务，以及创建新的任务从处于发布状态的任务的结束 Offset 处开始读取数据。

使用该接口可以无缝衔接地处理 Schema 变更。

7.6.2 关闭 Supervisor

POST /druid/indexer/v1/supervisor/<supervisorId>/shutdown

调用该接口，Supervisor 会立即停止，同时使其管理的所有任务终止读取，进入 Segment 发布状态。

7.6.3 获取当前执行的 Supervisor

GET /druid/indexer/v1/supervisor

返回当前活跃的 Supervisor 列表。

7.6.4 获取 Supervisor 规范

GET /druid/indexer/v1/supervisor/<supervisorId>

获取指定 supervisorId 的正在使用的规范。

7.6.5 获取 Supervisor 的状态报告

GET /druid/indexer/v1/supervisor/<supervisorId>/status

获取其管理的所有任务的状态报告。

7.6.6 获取所有 Supervisor 的历史

GET /druid/indexer/v1/supervisor/history

获取所有 Supervisor 的历史规范列表，包括当前使用的。

7.6.7 获取 Supervisor 的历史

GET /druid/indexer/v1/supervisor/<supervisorId>/history

获取指定 Supervisor 的历史规范列表。

7.7 最佳实践

7.7.1 容量规划

Kafka 索引任务运行在中间管理者上，因此其容量的使用上限受中间管理者集群规模的限制。在生产实践中，需要保障有充足的 Worker 容量。Worker 容量会被所有类型的任务共享使用，例如实时任务、合并任务等。所以在规划容量时，需要考虑所有的索引任务。如果超出容量限制以后，Kafka 索引任务在队列中等待，直到有可用的 Worker。这样有可能导致查询时部分结果延迟，但不会导致数据丢失（假设在任务执行之前，Kafka 不会清理任务起始 Offset 的数据）。

处于读取状态的任务数量由 taskCount 和 replicas 来控制。一般情况下，处于读取状态的任务总数为 $\text{replicas} \times \text{taskCount}$ ，但也有例外的情况，例如 taskCount 大于 Partition 的数量 {numKafkaPartitions}，在这种情况下只需要 {numKafkaPartitions} 个任务。当任务处于发布状态以后，又会有 $\text{replicas} \times \text{taskCount}$ 数量的新任务创建，因此要保障处于读取状态和发布状态的任务并发运行，所以需要的最小容量为： $\text{workerCapacity} = 2 \times \text{replicas} \times \text{taskCount}$ 。

一个读取状态的任务对应一个发布状态的任务，这是非常理想的状态。在有些情况下，会有多个发布状态的任务，这种情况在发布执行的时间（生成 Segment、推送到深度存储中，以及等待被历史节点加载）大于 taskDuration 时出现。为了减少容量的使用，最好将 taskDuration 调整为足够长，给发布执行充足的时间。

7.7.2 Supervisor 的持久化

当 Supervisor 规范通过 POST /druid/indexer/v1/supervisor 提交以后，会存储到元数据存储数据库中。

当 Overlord 成为 Leader 以后, 无论是初始执行还是其他 Overlord 失败, 它都会为元数据存储数据库中的每个 Supervisor 规范派生创建一个 Supervisor。Supervisor 会探索正在运行的任务, 如果任务和 Supervisor 规范兼容, 它就接管这个任务。摄入规范和 Partition 分配不同都会导致不兼容, Supervisor 会杀掉这样的任务, 同时创建新的任务集合。

7.7.3 Schema 的配置与变更

当 Schema 或者配置变更以后, 只需要重新提交新的 Supervisor 规范创建 Supervisor, 假设存在正在运行的 Supervisor, 则会将其停止, 同时使其管理的所有任务结束读取状态, 进入发布状态, 然后创建新的任务继续从处于发布状态的结束 Offset 处读取数据, 这样可以不需要等待, 实现无缝变更。

7.8 小结

本章介绍了 Druid 的几个高级功能, 其中直方图和 DataSketch 在应用中还是比较常见的, 特别是用于独立访问用户的计算, 可以采用 HyperLogLog 或 DataSketch 的方法。Kafka 索引服务也是对当前时间窗口模式和 Kafka 限制的一些突破, 帮助实现更加可靠、灵活的数据摄入。

第8章

核心源代码探析

本章介绍 Druid 的代码结构，以及如何编译项目。重点分析 Druid 的最基础数据结构，包括 Column、Index 和 Segment 等，还包括数据装载和持久化的基本流程，而后讨论 Druid 查询语句的底层实现，最后简单介绍 Coordinator 的实现原理。

Druid 的社区非常活跃，项目也处于高速发展阶段，代码不断持续优化，但是基本的数据结构还保持相对稳定，查询过程的变化也较少。了解这些数据结构和原理，可以更好地理解 Druid 的优势和局限性。

本章以 Druid 0.9 版本为基础，介绍 Druid 的一些底层设计理念，也有部分代码来自于 0.9.1 版本，已经特别注明。

8.1 如何编译 Druid 代码

Druid 项目需要有两个依赖软件：一个是 JDK；另一个是 Maven。JDK 需要版本 7 以上，建议使用 JDK 8 版本，因为社区开始讨论关于逐渐取消 Java 7 的支持。Maven 需要版本 3 以上。

配置好 JDK 和 Maven 之后，通过以下命令就可以下载和编译 Druid 项目了。

```
git clone git@github.com:druid-io/druid.git
cd druid
mvn clean package
```

编译之后，发布文件将位于 `distribution/target/druid-VERSION-bin.tar.gz`。

8.2 Druid 项目介绍

通过 IntelliJ 引入 Druid 的项目列表如图 8-1 所示。

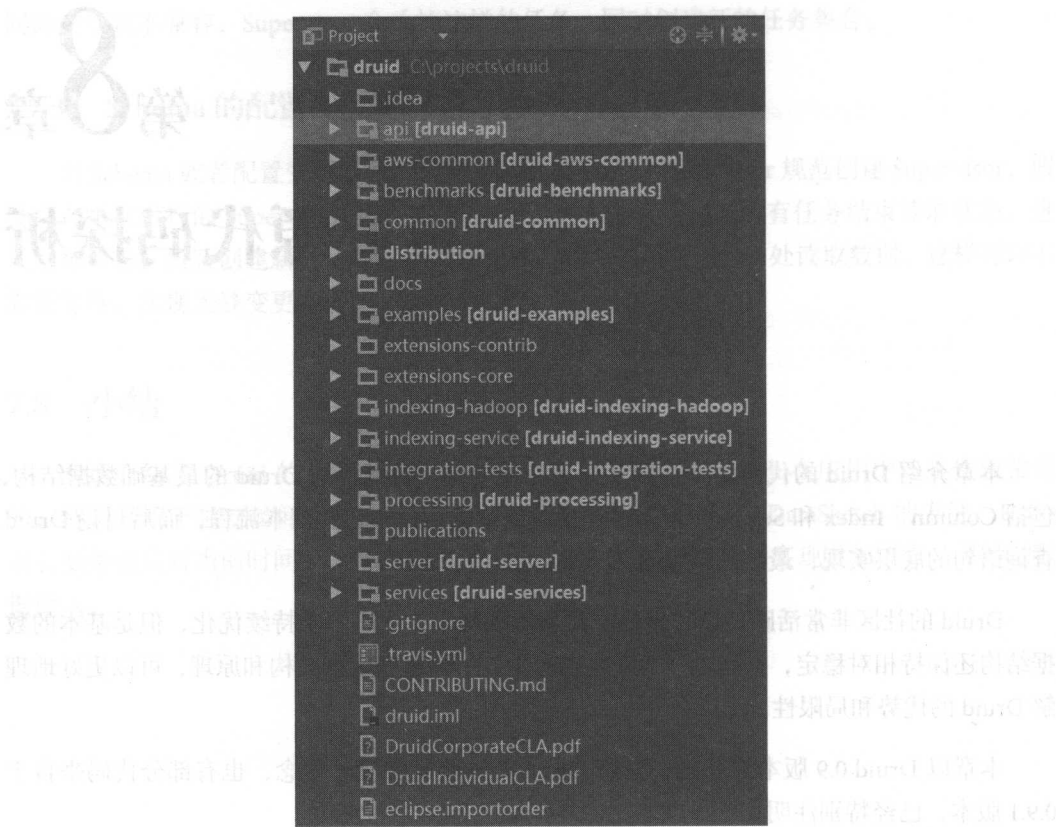


图 8-1 Druid 项目列表

这些项目简单介绍如下：

项目目录	功能	java 文件数量
核心项目		
processing	内部数据结构和处理	706
server	组合各种功能类，形成服务	562
indexing-service	索引服务，需要接收各种请求	194
services	构建物理服务，包括命令行解析	41
api	对外访问接口，解析各种访问	110

续表

项目目录	功能	.java 文件数量
扩展接口		
extensions-core	核心扩展包，包括与 Kafka, HDFS, S3 的集成	163
indexing-hadoop	与 Hadoop 索引相关的代码	78
aws-common	与 AWS 集成相关，包括账号权限管理等	5
extensions-contrib	社区开发者可以在这个目录下增加扩展功能，但这个包的内容不会放入最后的 Druid 发行版中	86
文档相关		
docs	项目文档	—
publications	一些相关论文，包括 KDD 论文	—

如果想快速浏览代码，可以从如下几个文件开始。

项目目录	功能	代码量 (.java)
核心项目		
Column.java	列	\$druid\processing\src\main\java\io\druid\segment\column\Column.java
Segment.java	Segment	\$druid\processing\src\main\java\io\druid\segment\Segment.java
IndexIO.java	装入 Index 文件	\$druid\druid\processing\src\main\java\io\druid\segment\IndexIO.java
IndexMerger.java	持久化 Index	\$druid\druid\processing\src\main\java\io\druid\segment\IndexMerger.java
IncrementalIndex.java	增量索引	\$druid\processing\src\main\java\io\druid\segment\incremental\IncrementalIndex.java
Main.java	服务程序入口	\$druid\services\src\main\java\io\druid\cli\Main.java

8.3 索引结构模块和层次关系

数据库的数据结构往往比较复杂。有些数据结构表述逻辑视图，有的表述具体实现等，有的表述存储视图，有的表述内存结构视图，这些数据结构之间通常使用以下几种关系和模式设计。

- 继承关系：父类通常为接口，子类继承父类，完成实现具体功能。
- 包含关系：一个对象包含几个其他的对象，实现一些组合功能。
- 适配器（Adapter）设计模式：该设计模式在数据基础结构中大量应用，比如我们已经有内存中的 Index 存储，为了支持查询功能，也需要提供适配器，将面向存储的 Index 适配成有查询能力的对象。
- 修饰者（Decoration）设计模式：为了增强一些类的某些功能，又不失去灵活性，通过包含对象方式继承某类大部分功能，同时也灵活地附加一些额外的功能。

8.4 Column 结构

Column 的结构如图 8-2 所示。

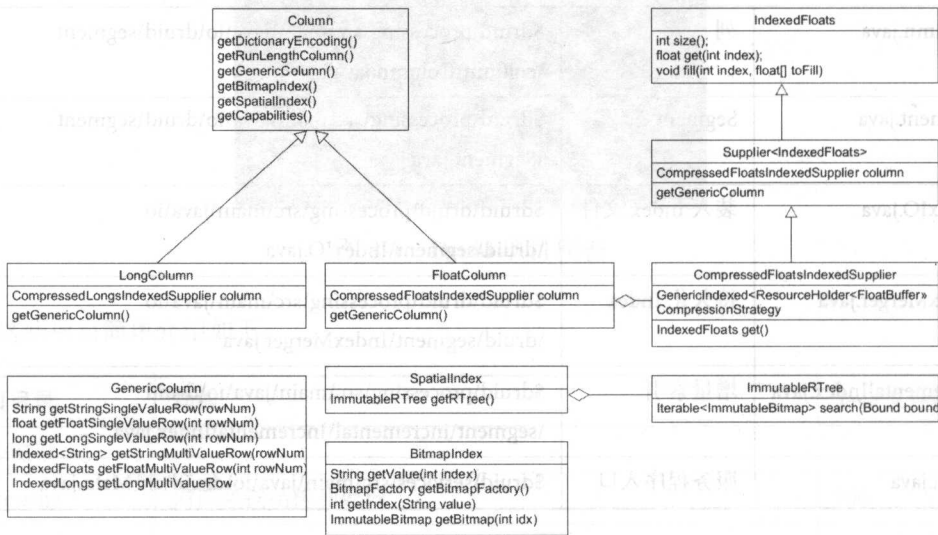


图 8-2 Column 的结构

Column 是基础列的基础接口。继承它的类包括 LongColumn、FloatColumn 和 SimpleColumn。以具体的 Float 类型为例，FloatColumn 继承了 Column，它的具体实现是使用 Com-

pressedFloatIndexSupplier。而 CompressedFloatIndex 能够返回 IndexFloats，它可以认为是一个 Floats 的列表结构。

1. Column

Column 是 Druid 最基础的数据结构，代码位于 \processing\src\main\java\io\druid\segment\column\Column.java。它是一个接口并没有提供任何通用的定义，而是提供了几种 Column 的类型，例如 FloatColumn、LongColumn 等。这个 Column 提供了一个非常抽象的结构。

```
public interface Column
```

```
{
```

```
    public static final String TIME_COLUMN_NAME = "__time";
```

```
    public ColumnCapabilities getCapabilities();
```

```
    public int getLength();
```

```
    public DictionaryEncodedColumn getDictionaryEncoding();
```

```
    public RunLengthColumn getRunLengthColumn();
```

```
    public GenericColumn getGenericColumn();
```

```
    public ComplexColumn getComplexColumn();
```

```
    public BitmapIndex getBitmapIndex();
```

```
    public SpatialIndex getSpatialIndex();
```

```
}
```

- Column 并没有提供一些通用的接口，而是提供了不同类型的 Column 的获取方法，具体 Column 的实现由每个返回的 Column 实体对象负责。
- GenericColumn 是一般的列，包括字符串（String）、浮点数（Float）和整数（Long）。
- DictionaryEncodedColumn 表示字典编码索引，Druid 中的字符串的列实际上使用的都是这种结构，在值的基数不大的情况下都可以使用这种模式。它提供了 getCardinality() 获取基数。
- RunLengthColumn 用于表示行程编码的列，尚未完全实现。这种压缩方式对于数据稀疏的列有较好的压缩率和访问速度。
- ComplexColumn 是一种复杂对象，常常用于一些扩展的数据类型，例如 HyperLoglog 和 Histogram 等。
- BitmapIndex 是 Druid 最核心的数据结构之一，它为列中的每一个值都创建一个 Bitmap，Bitmap 在内存中也会进行压缩，因此查询扫描时能够兼顾速度和内存大小。

2. FloatColumn

下面是一个 FloatColumn 例子，它继承了 AbstractColumn，AbstractColumn 继承了 Column。

```
public class FloatColumn extends AbstractColumn {
    private static final ColumnCapabilitiesImpl CAPABILITIES
        = new ColumnCapabilitiesImpl().setType(ValueType.FLOAT);

    private final CompressedFloatsIndexedSupplier column;

    public FloatColumn(CompressedFloatsIndexedSupplier column)
    {
        this.column = column;
    }

    @Override
    public ColumnCapabilities getCapabilities()
    {
        return CAPABILITIES;
    }

    @Override
    public int getLength()
    {
        return column.size();
    }

    @Override
    public GenericColumn getGenericColumn()
    {
        return new IndexedFloatsGenericColumn(column.get());
    }
}
```

从上面代码中，我们看出 FloatColumn 实际返回的是 IndexedFloatsGenericColumn()，这是一个浮点数的列。

3. BitmapIndex

```
public interface BitmapIndex
{
    public int getCardinality();
    public String getValue(int index);
    public boolean hasNulls();
    public BitmapFactory getBitmapFactory();
    public int getIndex(String value);
    public ImmutableBitmap getBitmap(int idx);
}
```

BitmapIndex 是 Druid 的核心结构，用于构造字符串类型列的索引。它为每个列值都创建一个 ImmutableBitmap。每个列值都有一个索引号 (idx)，所以内部通常使用 idx 获取 Bitmap。Bitmap 的对象由 BitmapFactory 生成。

8.5 Segment

Column 用于管理单列，Segment 就用于管理一组列，包括 Dimension 和 Metric。Segment 提供了两类接口：一类是 QueryableIndex；另一类是 StorageAdapter。QueryableIndex 是面向查询的数据接口，它提供了访问每一列的能力。StorageAdapter 具有游标 (Cursor) 功能，它提供了查询每行数据的能力。QueryableIndex 在一定方式下可以转换成 StorageAdapter 接口。Segment 的结构如图 8-3 所示。

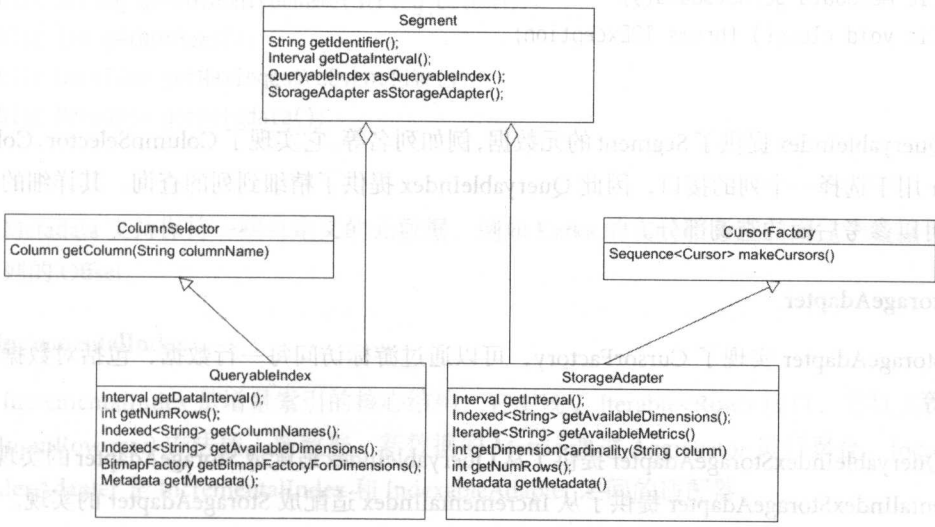


图 8-3 Segment 的结构

```
public interface Segment extends Closeable
{
    public String getIdentifier();
    public Interval getDataInterval();
    public QueryableIndex asQueryableIndex();
    public StorageAdapter asStorageAdapter();
}
```

1. QueryableIndex

QueryableIndex 提供了访问每一列的能力，支持针对某些列的查询。

```
public interface ColumnSelector
{
    public Column getColumn(String columnName);
}

public interface QueryableIndex extends ColumnSelector, Closeable
{
    public Interval getDataInterval();
    public int getNumRows();
    public Indexed<String> getColumnNames();
    public Indexed<String> getAvailableDimensions();
    public BitmapFactory getBitmapFactoryForDimensions();
    public Metadata getMetadata();
    public void close() throws IOException;
}
```

QueryableIndex 提供了 Segment 的元数据，例如列名等。它实现了 ColumnSelector, ColumnSelector 用于选择一个列的接口，因此 QueryableIndex 提供了精细到列的查询。其详细的使用方式可以参考后面的查询部分。

2. StorageAdapter

StorageAdapter 实现了 CursorFactory，可以通过游标访问每一行数据，包括对数据进行过滤等。

QueryableIndexStorageAdapter 提供了从 QueryableIndex 适配成 StorageAdapter 的实现；IncrementalIndexStorageAdapter 提供了从 IncrementalIndex 适配成 StorageAdapter 的实现。在转化过程中，构建一个游标，并且将列中的每一个值都加入到 Row 中。

```

public interface CursorFactory
{
    public Sequence<Cursor> makeCursors(Filter filter,
                                         Interval interval,
                                         QueryGranularity gran,
                                         boolean descending);
}

public interface StorageAdapter extends CursorFactory
{
    public String getSegmentIdentifier();
    public Interval getInterval();
    public Indexed<String> getAvailableDimensions();
    public Iterable<String> getAvailableMetrics();

    public int getDimensionCardinality(String column);
    public DateTime getMinTime();
    public DateTime getMaxTime();
    public Comparable getMinValue(String column);
    public Comparable getMaxValue(String column);
    public Capabilities getCapabilities();
    public ColumnCapabilities getColumnCapabilities(String column);

    public String getColumnTypeName(String column);
    public int getNumRows();
    public DateTime getMaxIngestedEventTime();
    public Metadata getMetadata();
}

```

Metadata 支持保持一些自定义的元数据，例如 Kafka 的实时服务就利用这个数据接口存储队列的 Offset。

3. IncrementalIndex

IncrementalIndex 是增量索引的核心结构，它实现了 Iterable<Row> 接口，并且支持通过 add(InputRow row) 方法插入新数据，新数据的 Metric 通过 Aggregator 进行聚合。IncrementalIndexAdapter 是 IncrementalIndex 和 IndexableAdapter 之间的适配器。

对于数据插入部分，可以参考 **Incremental** 的如下代码片段，其中包括聚合器的使用。

```
protected Integer addToFacts(
    AggregatorFactory[] metrics,
    boolean deserializeComplexMetrics,
    boolean reportParseExceptions,
    InputRow row,
    AtomicInteger numEntries,
    TimeAndDims key,
    ThreadLocal<InputRow> rowContainer,
    Supplier<InputRow> rowSupplier) throws IndexSizeExceededException
{
    final Integer priorIndex = facts.get(key);
    Aggregator[] aggs;

    if (null != priorIndex) {
        aggs = concurrentGet(priorIndex);
    } else {
        aggs = new Aggregator[metrics.length];
        rowContainer.set(row);
        for (int i = 0; i < metrics.length; i++) {
            final AggregatorFactory agg = metrics[i];
            aggs[i] = agg.factorize(selectors.get(agg.getName()));
        }
        rowContainer.set(null);

        final Integer rowIndex = indexIncrement.getAndIncrement();
        concurrentSet(rowIndex, aggs);

        // Last ditch sanity checks
        if (numEntries.get() >= maxRowCount && !facts.containsKey(key)) {
            throw new IndexSizeExceededException("Maximum number of rows [%d] reached",
                maxRowCount);
        }
        final Integer prev = facts.putIfAbsent(key, rowIndex);
        if (null == prev) {
            numEntries.incrementAndGet();
        } else {

```

```
// We lost a race
aggs = concurrentGet(prev);
// Free up the misfire
concurrentRemove(rowIndex);
// This is expected to occur ~80% of the time in the worst scenarios
}
}

rowContainer.set(row);

for (Aggregator agg : aggs) {
    synchronized (agg) {
        try {
            agg.aggregate();
        }
        catch (ParseException e) {
            // "aggregate" can throw ParseExceptions if a selector expects something but
            // gets something else.
            if (reportParseExceptions) {
                throw new ParseException(e, "Encountered parse error for aggregator[%s]",
                    agg.getName());
            } else {
                log.debug(e, "Encountered parse error, skipping aggregator[%s].", agg.
                    getName());
            }
        }
    }
}

rowContainer.set(null);
return numEntries.get();
}
```

在聚合过程中使用了 `aggregate`，注意 `aggregate` 并没有携带任何参数，这里利用了 `Thread-Local` 的特点。

Aggregator 是非常有意思的对象，用于聚合 **Metric**，提供了 **aggregate()** 和 **get()** 方法，不带任何参数，这里假设所有的参数都已经在环境中设置好，或者已经在构造函数中创建。它的内部实现使用了 **FloatColumnSelector** 和 **Offset**。

Aggregator 被认为是一种闭包（**Closure**）的设计，在多次调用中是有状态的模式，它可以由 **AggregatorFactory** 创建对象。**BufferAggregator** 是类似的聚合器，它将 **Metric** 直接聚合成 **ByteBuffer**，效率会更高一些。

```
public interface Aggregator {  
    void aggregate();  
    void reset();  
    Object get();  
    float getFloat();  
    String getName();  
    void close();  
    long getLong();  
}
```

另外，在聚合过程中使用了 **ThreadLocal** 技术，它为 **row** 数据创建了 **ThreadLocal** 变量：它为每一个线程分别创建一个独立的变量副本，而不会影响其他线程所对应的副本，提高了访问的并发度。

```
ThreadLocal<InputRow> rowContainer,  
...  
rowContainer.set(row);  
...//Aggregate row;  
rowContainer.set(null);
```

4. 装载索引文件

基本类图关系，**IndexIO** 提供装载索引文件功能：**LoadIndex(File DIR)**，返回 **QueryableIndex** 对象；其大部分实现由 **DefaultIndexIOHandler** 完成。**IndexIO** 实现类图如图 8-4 所示。

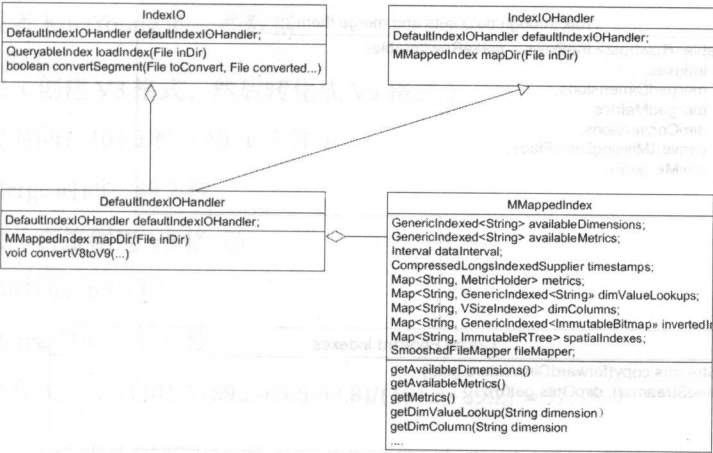


图 8-4 IndexIO 实现类图

装载索引文件的过程如图 8-5 所示。

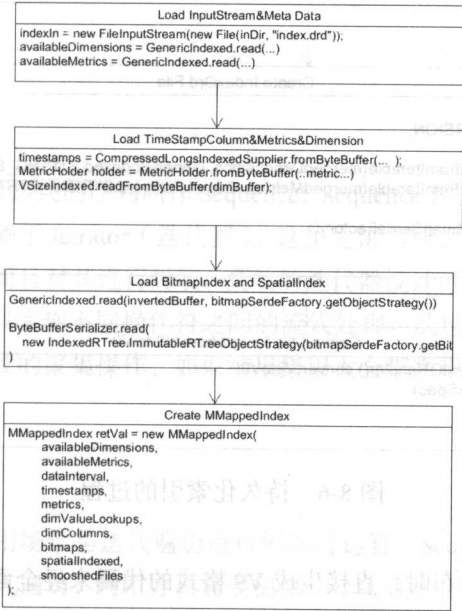


图 8-5 装载索引文件的过程

5. 持久化索引

IndexMerger 负责索引的持久化，整个过程如图 8-6 所示。

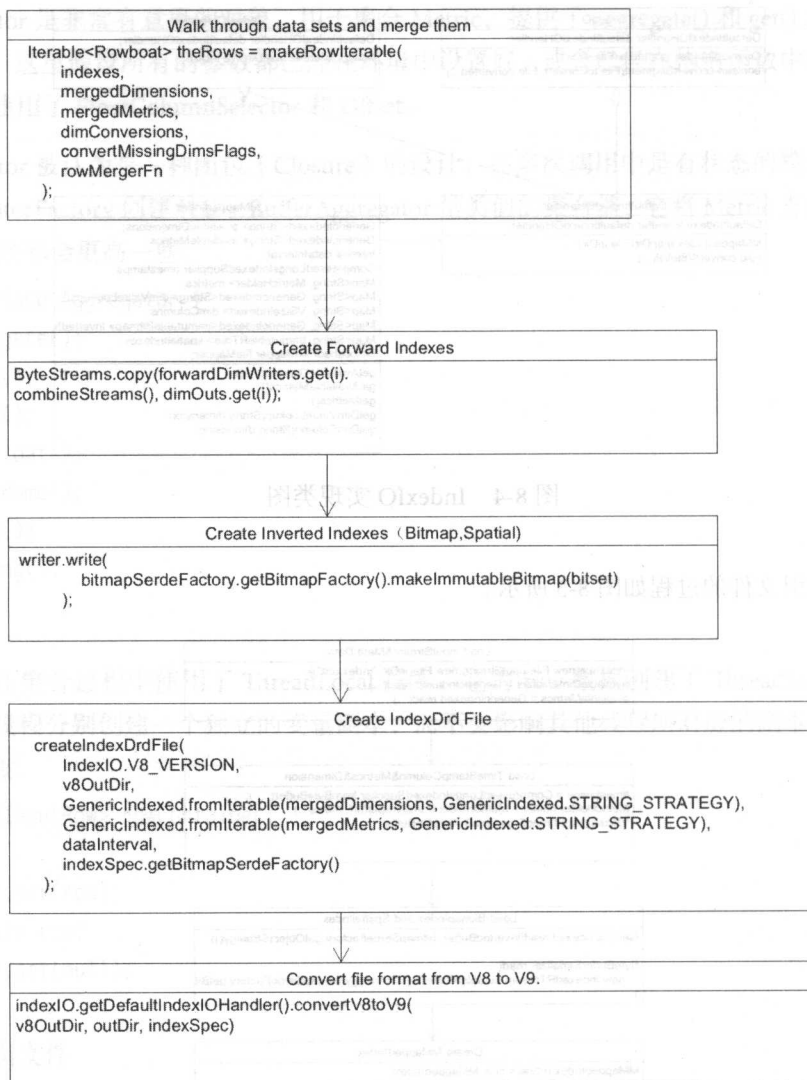


图 8-6 持久化索引的过程

由于 Druid 0.9 版本发布时，直接生成 V9 格式的代码未经全面测试，因此在 0.9 版本中的实现都是先生成 V8 格式，最后一步转化成 V9 格式。2016 年 6 月发布的 Druid 0.9.1 版本，开始支持直接生成 V9 格式的索引文件，提高了性能。下面是开发中的一些性能数据。

来自开发者 KurtYoung 的一些数据。

- 旧方法（创建 V8 格式，然后转化成 V9 格式）
 - 总时间：101.3 秒（10 个文件）
 - Merge 时间：89.2 秒
- 新方法（直接创建 V9 格式）
 - 总时间：65.6 秒
 - Merge 时间：53.8 秒
- 性能提升 37.3% $((101.3+89.2-65.6-53.8)/(101.3+89.2) = 37.3\%)$

直接生产 V9 格式代码可以查看 `$\druid\processing\src\main\java\io\druid\segment\Index-MergerV9.java`。

8.6 Query 模块

8.6.1 基础组件

1. Sequence

Druid 设计了一种可迭代的序列叫作 Sequence，Sequence 其实是对 Iterator 通用操作的高级封装，其底层还是依赖于 Iterator（迭代器）。这里先讲为什么要使用迭代器，后续再讲为什么不直接使用迭代器而且对其进行封装。高效的迭代器设计可以实现只扫描一次表就能完成操作，使用迭代器可以实现不同操作符之间的流式处理。试想使用 List 存储扫描索引以后得到的结果集传递给后续的聚集操作，如果结果集很大会带来很大的内存开销，同时会造成扫描遍历多次。

2. Sequence 实现

迭代器最典型的应用场景是迭代遍历进行操作符运算。Sequence 采用控制反转的方式，在 Sequence 内部完成迭代遍历，调用方只需要提供一个实现操作符逻辑的函数，在遍历过程进行回调即可。

这样设计的好处是 Sequence 能够对资源进行管理，它可以在遍历完成以后强制调用 close 方法完成对资源的关闭操作。控制反转的缺点是不能把迭代器中的数据暴露给用户使用。为了解决这个问题，可以将 Sequence 转换成 Yelder 对象。Yelder 的功能类似于 Python 的 yield 方法，可以在遍历过程中中断，它会保存执行的状态，下次执行时再从中断处开始。但 Yelder

不会提供类似于 `Sequence` 的资源管理功能，需要调用方显式地调用 `close` 方法。`Sequence` 和 `Yielder` 的功能看起来很神奇，接下来从代码层面剖析其是如何实现的。先看 `Sequence` 接口，它只提供了如下两个方法。

```
public <OutType> OutType accumulate(OutType initValue,
                                     Accumulator<OutType, T> accumulator);

public <OutType> Yielder<OutType> toYielder(OutType initValue,
                                             YieldingAccumulator<OutType, T> accumulator);
```

`accumulate` 方法的功能是通过控制反转的方式完成聚合运算。调用该方法时需要传入一个 `Accumulator`（累积器）。`Accumulator` 是封装回调函数的接口，把原来在迭代过程中进行聚合运算的逻辑抽取到 `accumulate` 方法中回调执行。

```
public interface Accumulator<AccumulatedType, InType>
{
    public AccumulatedType accumulate(AccumulatedType accumulated,
                                       InType in);
}
```

该方法接收两个参数，参数类型通过泛型的方式设定。第一个参数 `accumulated` 保存聚合运算的结果，在调用时作为参数传入，计算完成以后将该参数作为结果返回，并在下一次迭代时作为参数传入，循环执行直到迭代结束。具体过程请参考如下代码。

```
OutType retVal = initValue;
while (!accumulator.yielded() && iter.hasNext()) {
    retVal = accumulator.accumulate(retVal, iter.next());
}
```

第二个参数 `in` 是迭代器的下一个元素的值。听起来比较抽象，下面通过一个示例来说明。假设有一个计算所有元素的总和的场景，我们采用 `Sequence` 来实现。首先实现执行求和逻辑的 `Accumulator`，`in` 这个参数必须是 `Integer` 类型，`accumulated` 也设置为 `Integer` 类型。代码如下：

```
new Accumulator<Integer, Integer>() {
    @Override
    public Integer accumulate(Integer accumulated, Integer in) {
        return accumulated+in;
    }
};
```

第二个方法 `toYielder` 的功能是将 `Sequence` 转换成一个 `Yielder`。`Yielder` 对象可以看作是一个链表，调用 `Yielder` 的 `get` 方法获取当前头元素的值，通过调用 `next` 方法获取下一个 `Yielder` 对象。在 `toYielder` 方法中需要传入一个 `YieldingAccumulator`，它和 `Yielder` 协同工作实现 Java 语言中的中断/延续执行。`YieldingAccumulator` 添加了 `yield` 标志，`yield` 标志的初始值为 `false`，调用 `yield` 方法以后将该标志设置为 `true`，它的作用是退出当前的遍历迭代过程，并将累积器中的值赋给当前 `Yielder`。下面实现一个 `YieldingAccumulator`，目标是获取 `Sequence` 的每个整数元素，通过 `Yielder` 的接口来实现 `Iterator` 的 `next` 方法。

```
new YieldingAccumulator<Integer, Integer>() {  
    @Override  
    public Integer accumulate(Integer accumulated, Integer in) {  
        yield();  
        return in;  
    }  
};
```

奥秘就在第4行 `yield` 方法的调用上，退出当前的迭代，这样每执行一次迭代就退出并返回当前值一次，就实现了类似于 `Iterator` 的 `next` 方法。`BaseSequence` 是 `Sequence` 的基本实现。构建 `BaseSequence` 需要传入自定义的 `IteratorMaker` 对象，`IteratorMaker` 用来创建 `Iterator` 以及销毁 `Iterator` 背后的资源，并在其基础上采用装饰器模式衍生出很多不同功能的 `Sequence`。`Sequences` 是一个工具类，提供了一些对常用 `Sequence` 的封装。我们来看一下 `Sequences` 的一些常用方法。

- `simple` 方法，传入一个实现 `Iterable` 接口的对象，返回 `BaseSequence`。
- `concat` 方法，用于把多个 `Sequence` 合并成一个，为了减少内存的使用，并不会把多个 `Sequence` 中的元素复制到一个新的 `Sequence`，而是在执行 `accumulate` 方法时将多个 `Sequence` 的累积结果合并在一起。
- `map` 方法，最常见的方法，类似函数式编程中的 `map` 函数，在执行 `accumulate` 方法时，在调用转换函数以后再进行聚合操作。下面我们通过示例讲解具体使用方法。假设有一个 `Sequence` 对象保存学生的成绩，现在需要将每个学生的成绩提高5分，可以通过下面代码来实现。

```
Function<Integer, Integer> fn = new Function<Integer, Integer>()  
{  
    public Integer apply(Integer input)  
    {
```

```
return input + 5;
};
Sequences.map(source, fn);
```

- **filter** 方法,其功能是在执行 **accumulate** 方法时根据传入的 **Predicate** 过滤,如果 **Predicate** 返回 **true** 则进行累积,返回 **false** 则抛弃。**Group By** 查询的 **Having** 就是用 **filter** 方法实现的。
- **withBaggage** 方法,将传入的 **Sequence** 和 **Closable** 转换成 **ResourceClosingSequence**,其功能是 **Sequence** 主动管理在使用过程中除底层迭代器以外的其他资源,例如从内存池申请的内存。可以自定义 **Closable** 在其 **close** 方法中实现资源的释放。在 **Sequence** 调用 **close** 方法时,同时调用 **Closable** 的 **close** 方法实现主动管理资源,优点是调用方不再关心资源管理,避免调用方忘记释放资源造成泄漏。
- **withEffect** 方法,在执行 **accumulate** 方法时异步执行某些逻辑,例如在 **CachingQueryRunner** 中异步地将 **Sequence** 中的元素收集到一个 **List** 中,待 **accumulate** 方法执行完成再进行缓存操作。
- **toList** 方法,转换成 **List**,将 **Sequence** 中的元素物化到 **List** 中。
- **sort** 方法,将 **Sequence** 中的元素按照指定的规则排序。需要注意的是,该方法会先将 **Sequence** 中的元素物化到 **List** 中,然后排序 **List** 中的元素,再转换成 **Sequence**。因为需要物化到 **List** 中,如果 **Sequence** 很大的话,需要注意内存的使用。

8.6.2 内存池管理

为了减轻 JVM 垃圾回收带来的性能波动,Druid 尽量使用堆外内存和系统内存。

- 使用临时文件,在索引创建和合并过程中,中间临时结果会占用大量的内存。为了减少 JVM 内存的使用,采用临时文件,通过文件 IO 的方式,巧妙地利用内核的 **Page Cache** (页面缓存)。为了提升 IO 的性能,Linux 操作系统增加了 **Page Cache**,文件 IO 的写操作会写到 **Page Cache** 中后立即返回,内核会定时将脏页也就是还没有写到磁盘的页刷到磁盘中,但并不会删除相应的 **Page Cache**,以便读取时快速访问。Apache **Kafka** 也正是利用了文件 IO 的这种特性。
- **MMap** 在 Java 中通过调用 **ByteBuffer** 的 **get/put** 接口来实现文件的读写,它也是依赖于内核的 **Page Cache**。它最大的优势是直接操作内核的内存,减少一次内存复制。

- DirectBuffer, 在 Druid 中 DirectBuffer 的使用场景如下。
 - 在索引创建过程中按块压缩/编码, LZ4 压缩除外, 默认块大小为 64KB。
 - 在索引查询过程中按块解压缩/解码, 存放解压缩以后的数据, 默认块大小为 64KB。
 - 在查询过程中存放中间结果集。
 - Group By 查询在上下文中设置 “userOffHeap=true”, 则使用 DirectBuffer 存放计算结果集。

Druid 采用固定分区的内存池。固定分区的优点是足够简单, 缺点是每次申请分配固定大小的内存, 容易造成内存碎片。但根据上述的使用场景, 编码/解码、压缩/解压缩都是按块操作的, 所以固定分区非常切合这种场景。Group By 查询需要根据维度拉取原始数据, 然后在内存中进行聚合操作。默认的内存增量索引使用 JVM 内存。为了减少大量创建临时对象, Druid 设计了临时计算结果集, 采用固定大小的堆外内存存储 Metric 的值, 先在临时计算结果集中聚合。Druid 会给每个处理线程从内存池中申请一块内存, 内存大小通过 `buffer.sizeBytes` 设置。Group By 查询采用固定大小的内存, 会造成内存浪费。处理线程每次只处理一个 Segment 分片, 为了兼顾处理较大的 Segment 分片, 我们一般将内存块的大小设置大一些。这样处理较小的 Segment 分片就会造成内存浪费。

8.6.3 查询流程概览

Druid 查询的整体流程如图 8-7 所示。

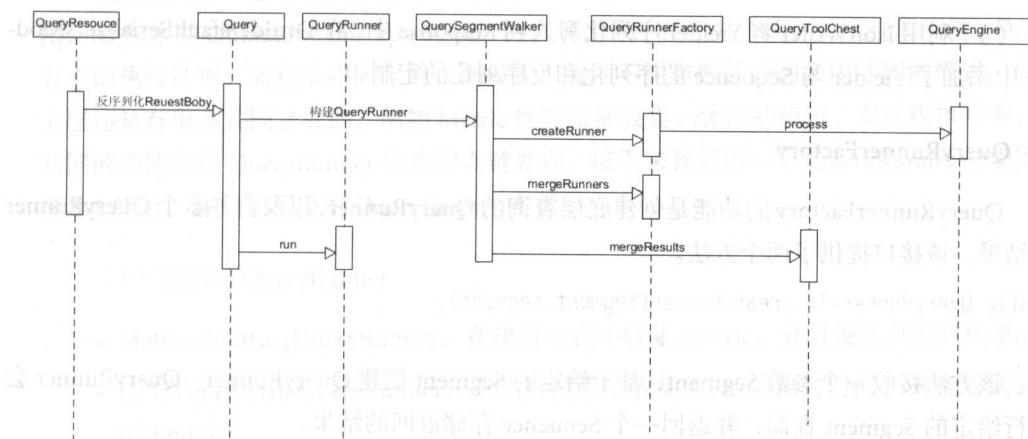


图 8-7 Druid 查询整体流程图

下面是 Druid 查询过程中的关键组件。

- QueryResource, 查询入口。
- QuerySegmentWalker, 根据查询的 interval 或者 Segment 构建 QueryRunner。
- QueryRunnerFactory, 负责构建底层查询的 QueryRunner, 以及并发执行 QueryRunner, 然后合并。
- QueryRunner, QueryRunner 采用调用链和装饰器模式实现, 每个 QueryRunner 负责执行查询链的一段逻辑。
- QueryToolChest, 辅助创建 QueryRunner, 用于合并结果以及后处理等。
- QueryEngine, 查询引擎, 查询逻辑的核心实现类。每种类型的查询都有一个相应的引擎。

1. QueryResource

QueryResource 是查询的入口, Druid 启动时将 “druid/v2” 的 URL 绑定在 QueryResource 上。其执行流程如下:

(1) 通过 Jackson Json 的 ObjectMapper 把参数反序列化成 Query 对象。

(2) 调用相应的 QuerySegmentWalker 构建 QueryRunner, 然后执行返回 Sequence。

(3) 将 Sequence 转换成 Yelder, 这个 Yelder 每执行一次返回一个元素, 类似于迭代器, 方便序列化。

(4) 利用 JsonWriter 将 Yelder 序列化写入到 Response 中。在 DruidDefaultSerializersModule 中添加了 Yelder 与 Sequence 的序列化和反序列化的定制。

2. QueryRunnerFactory

QueryRunnerFactory 的功能是创建底层查询的 QueryRunner, 以及合并多个 QueryRunner 的结果。该接口提供了两个方法。

```
public QueryRunner<T> createRunner(Segment segment);
```

该方法接收一个参数 Segment, 基于给定的 Segment 创建 QueryRunner。QueryRunner 会执行给定的 Segment 查询, 并返回一个 Sequence 存储返回的结果。

```
public QueryRunner<T> mergeRunners(ExecutorService queryExecutor,  
                                     Iterable<QueryRunner<T>> queryRunners);
```

大部分场景会查询多个 Segment, createRunner 方法根据每个给定的 Segment 分片创建 QueryRunner, mergeRunners 方法会将这些 QueryRunner 提交给 ExecutorService 并发执行, 最后合并其返回结果。Druid 设计了 ChainedExecutionQueryRunner 封装了上述逻辑。除了 Group By 查询和 SegmentMetadata 查询以外, 其他所有查询的 mergeRunners 方法实现如下面代码所示。

```
return new ChainedExecutionQueryRunner<Result<T>>(queryExecutor,  
                                                    queryWatcher,  
                                                    queryRunners);
```

后面会介绍 ChainedExecutionQueryRunner 的具体实现。不同类型的查询设计了其名称前缀的 QueryRunnerFactory。

- GroupByQueryRunnerFactory
- TimeseriesQueryRunnerFactory
- TopNQueryRunnerFactory
- SearchQueryRunnerFactory
- SelectQueryRunnerFactory
- TimeBoundaryQueryRunnerFactory

3. QueryRunner

QueryRunner 是封装执行查询的高级接口。QueryRunner 采用装饰器设计模式, 类似于 JDK IO 包中的 OutputStream/InputStream 的实现, 通过嵌套组合的方式实现职责链。Druid 没有查询执行计划, 而是采用固定的查询模式。每种不同类型的查询采用不同的模式, 但是其上层还是有很多共同之处的, 例如 Metric 性能指标收集、缓存的使用、多线程执行等, 这些共同的功能交给 QueryRunner 构建职责链处理。接下来我们看一下 QueryRunner 的类图, 如图 8-8 所示, 并介绍重要的 QueryRunner 实现。

(1) 通用的 QueryRunner

- MetricsEmittingQueryRunner, 在执行过程中收集 Metric, 并且发送到配置的 Emitter。
- CPUTimeMetricQueryRunner, 在执行过程中收集 CPU 的执行时间, 并且发送到配置的 Emitter。
- FinalizeResultsQueryRunner, 将复杂对象的 Metric 转换成数值类型。
- BySegmentQueryRunner, 用于调试, 在结果集中添上 Segment 的信息。

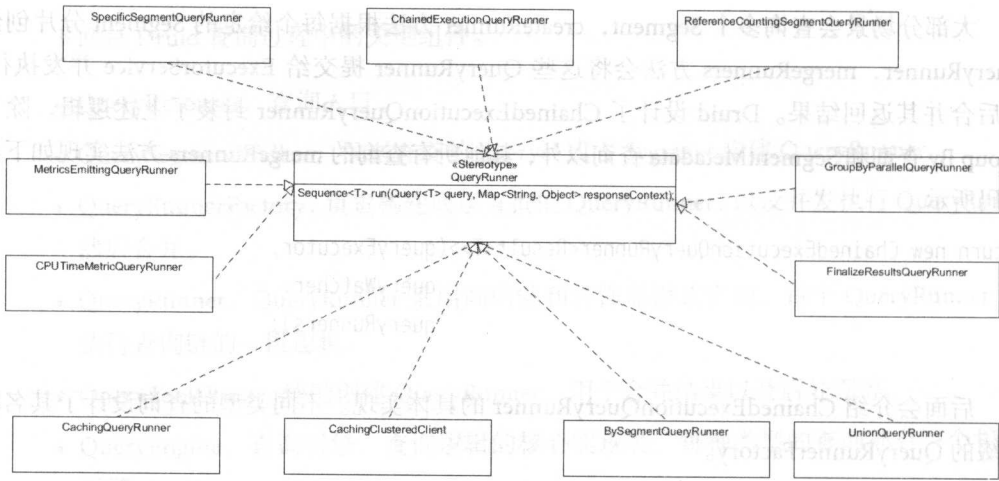


图 8-8 QueryRunner 类图

(2) Broker 用到的 QueryRunner

- CachingClusteredClient，它是 Broker 查询的核心类，根据时间线路由，分发查询请求到不同节点，然后汇总请求，并且将结果集缓存起来。Broker 中的查询缓存也在这个类中完成。
- DirectDruidClient，使用 NettyHttpClient 发起对历史节点或者实时节点（或者索引服务）的请求。
- UnionQueryRunner，它的功能是处理 Union 查询。
- IntervalChunkingQueryRunner，会将大跨度的查询按照 chunkPeriod 拆分成多个小段的查询并发执行，然后合并。

(3) 实时节点或历史节点用到的 QueryRunner

- ChainedExecutionQueryRunner，并发执行的核心类，它会将对不同 Segment 查询的 QueryRunner 提供给线程池去处理，最后合并结果。
- GroupByParallelQueryRunner，其功能类似于 ChainedExecutionQueryRunner。因为 Group By 查询模式特殊不同于其他类型的查询，需要根据维度分组聚合，所以要单独使用这个类来实现。
- CachingQueryRunner，添加缓存功能。
- SelectQueryRunner，调用底层的 Select 查询引擎，位于 SelectQueryRunnerFactory 的内部类中。

- GroupByQueryRunner, 调用底层的 Group By 查询引擎, 位于 GroupByQueryRunner-Factory 的内部类中。
- ReferenceCountingSegmentQueryRunner, 添加对 Segment 的引用计数逻辑, 防止正在使用的 Segment 被删除。
- SpecificSegmentQueryRunner, 该类的功能是将执行 QueryRunner 的当前线程的名字改成 “queryType_dataSource_interval” 的形式。

4. ChainedExecutionQueryRunner

Druid 的设计目标是面向用户的平台级产品, 为了提升查询性能, 以及解决高并发性问题, 采用 Scatter/Gather 模式, 将多个 Segment 的查询提交给线程池去并发执行, 然后阻塞地等待查询完成, 最后合并结果。前面提到除了 Group By 查询以外, 其他所有的查询均使用 ChainedExecutionQueryRunner 完成并发执行。Druid 通用查询模式如图 8-9 所示。

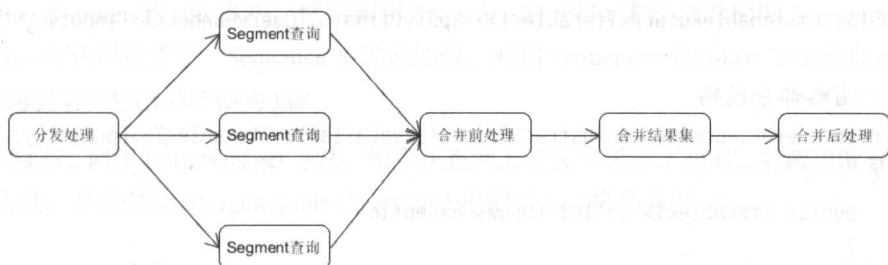


图 8-9 Druid 通用查询模式

分发处理的逻辑是在 QueryRunnerFactory 的 mergeRunners 方法中执行的。在 mergeRunners 方法中根据传入参数 exec 和 querRunners 构造 ChainedExecutionQueryRunner。这里以 TimeBoundary 查询为例, 代码见 druid-processing/io/druid/query/timeboundary/TimeBoundary-QueryRunnerFactory。

```

public QueryRunner<Result<TimeBoundaryResultValue>> mergeRunners(ExecutorService
    queryExecutor, Iterable<QueryRunner<Result<TimeBoundaryResultValue>>> queryRunners)
{
    return new ChainedExecutionQueryRunner<>(queryExecutor,
        queryWatcher,
        queryRunners);
}

```

下面我们根据源码分析 `ChainedExecutionQueryRunner` 的处理流程。

```

1  final int priority = BaseQuery.getContextPriority(query, 0);
2  final Ordering ordering = query.getResultOrdering();
3
4  return new BaseSequence<T, Iterator<T>>(
5      new BaseSequence.IteratorMaker<T, Iterator<T>>()
6      {
7          public Iterator<T> make()
8          {
9              ListenableFuture<List<Iterable<T>>> futures = Futures.allAsList(Lists.newArrayList(
10                  Iterables.transform(queryables
11                      new Function<QueryRunner<T>, ListenableFuture<Iterable<T>>>()
12                      {
13                          @Override
14                          public ListenableFuture<Iterable<T>> apply(final QueryRunner<T> input)
15                          {
16                              //省略部分代码
17                              return exec.submit(new AbstractPrioritizedCallable<Iterable<T>>(priority)
18                                  {
19                                      public Iterable<T> call() throws Exception
20                                      {
21                                          try {
22                                              Sequence<T> result = input.run(query, responseContext);
23                                              List<T> retVal = Sequences.toList(result, Lists.<T>newArrayList());
24                                              return retVal;
25                                          }
26                                          //省略部分不重要的代码
27                                      }
28                                  }
29                              );
30                              queryWatcher.registerQuery(query, futures);
31                          }
32                      }
33                      final Number timeout = query.getContextValue(QueryContextKeys.TIMEOUT, (Number) null);
34                      return new MergeIterable<>(
35                          ordering.nullsFirst(),
36                          timeout == null ?
37                          futures.get() :
38                          futures.get(timeout.longValue(), TimeUnit.MILLISECONDS)).iterator();
39                      }
40                  }
41              );
42          }
43      }
44  );

```

第9行, 利用 Iterables 的 transform 方法, 将 QueryRunner 转换为一组 ListenableFuture, 作为 List 添加到 Futures 的 allAsList 方法中。这里并没有使用 JDK 自带的 Future, 而是采用 Google 的 Guava 提供的 ListenableFuture, 因为使用 JDK 的 Future, 我们需要不断地轮询查看 List 中的每个 Future 是否完成, 而 ListenableFuture 通过添加 Listener 的方式, 一旦完成以后就触发告知监听者。Futures 的 allAsList 会创建一个 CombinedFuture 并返回, 它会监听传入的所有 ListenableFuture, 如果其执行完成, 则会将结果按照顺序放入 List 相应的位置, 调用它的 get 方法时, 会检查那些传入的 ListenableFuture 是否都完成, 完成则返回, 没有则等待。同时 allAsList 方法要求所有的 ListenableFuture 都执行成功才返回结果, 一旦任何一个失败, 就会返回失败。同时它会监听 cancel, 一旦调用它的 cancel 方法, 就会取消所有传入的 ListenableFuture。

从第17行开始, 构建一个 AbstractPrioritizedCallable 的匿名类, 这是带有优先级的 Callable, 其构造函数的参数 priority 表示执行的优先级, 它可以在查询上下文中设置。在 call 方法中执行 QueryRunner 的 run 方法, 然后调用 Sequence 的 toList 方法将返回的 Sequence 转换为 List, 这一行代码很重要, Sequence 是延迟加载, 调用 Sequences 的 toList 方法会真正调用底层的查询引擎并将结果转换为 List。

第34行, 调用 futures 的 get 方法, 如果在查询上下文中设置了超时, 则调用带有 timeout 的 get 方法。然后使用 MergeIterable 将 get 方法返回的 List 排序合并。

5. GroupByParallelQueryRunner

GroupByParallelQueryRunner 的功能以及实现和 ChainedExecutionQueryRunner 在整体架构上非常类似, 不同之处是 Group By 查询, 需要汇总每个 Segment 的列表, 然后在内存中聚合。这个过程听起来比较抽象, 下面我们通过一个实例来解释一下。假设一个 DataSource 有 A、B、C、D 四个维度和 sum 一个度量值。按照 A 分组统计 sum, 数据分布在 3 个 Segment 上。维度 A 的基数是 4, 分别是 A1、A2、A3 和 A4。其查询示意图如图 8-10 所示。

使用 ChainedExecutionQueryRunner 处理 Group By 查询, 它会将每个 Segment 查询返回的 Sequence 物化为 List, 然后交给 QueryToolChest, 在合并结果集时采用 IncrementalIndex 进行聚合。像 Timeseries 和 TopN 查询, List 的大小是可预测的, 内存的使用是可控的; 但对于 Group By 查询, List 的大小是由参与分组的维度组合以及维度的基数决定的, 如果维度的基数特别大以及维度的组合很多, 那么转换成的 List 会占用很多的内存。为了优化内存的使用, 从理论上讲, 根据 Group by 查询的特性, 将 Sequence 转换成 List 是多余的。优化方案是直接迭代 Sequence, 并将结果在同一 IncrementalIndex 累积聚合, 如图 8-11 所示。

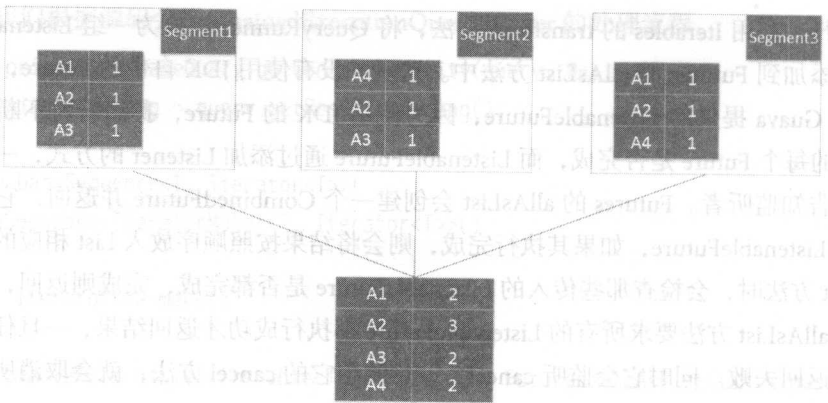


图 8-10 Group By 查询示意图

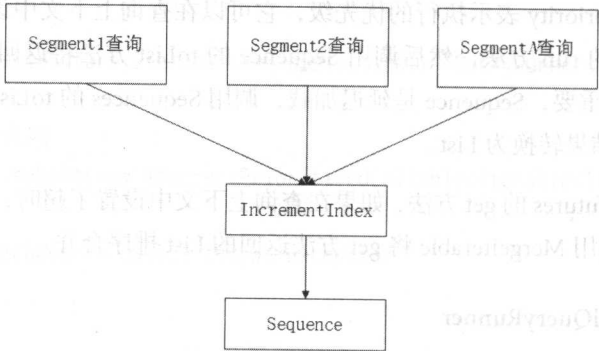


图 8-11 优化方案示意图

下面分析 GroupByParallelQueryRunner 的关键代码实现。

```
1 final GroupByQuery query = (GroupByQuery) queryParam;
2 final Pair<IncrementalIndex, Accumulator<IncrementalIndex, T>> indexAccumulatorPair =
  GroupByQueryHelper.createIndexAccumulatorPair(
3   query,
4   configSupplier.get(),
5   bufferPool);
6 final Pair<Queue, Accumulator<Queue, T>> bySegmentAccumulatorPair = GroupByQueryHelper.
  createBySegmentAccumulatorPair();
7 final boolean bySegment = BaseQuery.getContextBySegment(query, false);
8 final int priority = BaseQuery.getContextPriority(query, 0);
9
10 //省略部分代码
```

```
11 public Void call() throws Exception
12 {
13     try {
14         if (bySegment) {
15             input.run(queryParam,responseContext).accumulate(bySegmentAccumulatorPair.lhs,
16                 bySegmentAccumulatorPair.rhs);
17         } else {
18             input.run(queryParam,responseContext).accumulate(indexAccumulatorPair.lhs,
19                 indexAccumulatorPair.rhs);
20         }
21     }
22     return null;
23 } catch (...) {
24     ...
25 }
26 }
```

与 ChainedExecutionQueryRunner 的框架代码一致，不同的是 Callable 中的实现逻辑。

第 15 行代码是根据 bySegment 判断，bySegment 标志用于调试，如果 bySegment 为 true，则不进行聚合，直接将明细数据返回。两者的不同之处是收集的容器和 Accumulator。

第 17 行代码是普通非 Debug 模式调用的，它使用 Pair 对象保存容器和 Accumulator，lhs 是容器对象 IncrementalIndex，rhs 是 Accumulator，是一个匿名类，它实现的逻辑是将 Sequence 中的 Row 添加到 IncrementalIndex 中聚合。

第 2 行代码是 indexAccumulatorPair 的构建，这块代码是 Group By 查询的核心逻辑之一，抽取到 GroupByQueryHelper 类中实现。

```
1 final QueryGranularity gran = query.getGranularity();
2 final long timeStart = query.getIntervals().get(0).getStartMillis();
3 final long granTimeStart = gran.iterable(timeStart, timeStart + 1).iterator().next();
4 final List<AggregatorFactory> aggs = Lists.transform(
5     query.getAggregatorSpecs(),
6     new Function<AggregatorFactory, AggregatorFactory>()
7     {
8         @Override
9         public AggregatorFactory apply(AggregatorFactory input)
10         {
11             return input.getCombiningFactory();
```

```

12     }
13 }
14 );
15 final List<String> dimensions = Lists.transform(
16     query.getDimensions(),
17     new Function<DimensionSpec, String>()
18     {
19         @Override
20         public String apply(DimensionSpec input)
21         {
22             return input.getOutputName();
23         }
24     }
25 );
26 final IncrementalIndex index;
27 final boolean sortResults = query.getContextValue(CTX_KEY_SORT_RESULTS, true);
28 if (query.getContextValue("useOffheap", false)) {
29     index = new OffheapIncrementalIndex(
30         // use granularity truncated min timestamp
31         // since incoming truncated timestamps may precede timeStart
32         granTimeStart,
33         gran,
34         aggs.toArray(new AggregatorFactory[aggs.size()]),
35         false,
36         true,
37         sortResults,
38         Math.min(query.getContextValue(CTX_KEY_MAX_RESULTS, config.getMaxResults()), config.
39             getMaxResults()), bufferPool);
40 } else {
41     index = new OnheapIncrementalIndex(
42         // use granularity truncated min timestamp
43         // since incoming truncated timestamps may precede timeStart
44         granTimeStart,
45         gran,
46         aggs.toArray(new AggregatorFactory[aggs.size()]),
47         false,
48         true,
49         sortResults,

```

```
49     Math.min(query.getContextValue(CTX_KEY_MAX_RESULTS, config.getMaxResults()), config.  
        getMaxResults());  
50     );  
51 }  
52  
53 Accumulator<IncrementalIndex, T> accumulator = new Accumulator<IncrementalIndex, T>() {  
54     @Override  
55     public IncrementalIndex accumulate(IncrementalIndex accumulated, T in)  
56     {  
57         if (in instanceof MapBasedRow) {  
58             try {  
59                 MapBasedRow row = (MapBasedRow) in;  
60                 accumulated.add(  
61                     new MapBasedInputRow(  
62                         row.getTimestamp(),  
63                         dimensions,  
64                         row.getEvent()  
65                     )  
66                 );  
67             } catch (IndexSizeExceededException e) {  
68                 throw new ISE(e.getMessage());  
69             }  
70         } //省略部分代码  
71         return accumulated;  
72     }  
73 };  
74 return new Pair<>(index, accumulator);
```

为了便于理解源码,采用场景对照法,这里设计了一个 GroupBy 的查询场景,源自 Druid 官方文档中的 Group By 查询一节,这个查询是按照 country 和 device 两个维度进行分组,统计 total_usage 和 data_transfer,查询的 JSON 表达如下。

```
{  
    "queryType": "groupBy",  
    "dataSource": "sample_datasource",  
    "granularity": "hour",  
    "dimensions": ["country", "device"],  
    "aggregations": [  
        "total_usage",  
        "data_transfer"
```



```

    { "type": "longSum", "name": "total_usage", "fieldName": "user_count" },
    { "type": "doubleSum", "name": "data_transfer", "fieldName": "data_transfer" }
  ],
  "intervals": ["2012-01-01T00:00:00.000/2012-01-03T00:00:00.000"]
}

```

接下来对照源码来阐述上述查询的执行过程。

第 1 行是从 query 中获取 QueryGranularity，上述查询的 Granularity 是 day。

第 2 行是获取查询的时间区间的最小值，上述查询的时间区间的最小值是 2012-01-01T00:00:00.000。

第 3 行是获取 minTimestamp，如果添加到 IncrementalIndex 的 event 的时间戳小于 minTimestamp，则会抛出异常。

第 4~14 行是根据查询中的 aggregations 获取 AggregatorFactory，AggregatorFactory 是一个工厂类，用于创建相应的聚合器。上述查询中有两个聚合器，分别是 longSum 和 doubleSum。

第 15~25 行是获取用于分组的维度。值得注意的是，第 22 行使用的是 outputName，主要是为了兼顾抽取函数，如果没有使用抽取函数，outputName 则使用维度名。上述查询中的维度是 country 和 device，也就是说，按照 country 和 device 组合进行分组查询。

第 28~53 行是创建 IncrementalIndex，首先会根据查询上下文中的 useOffHeap 来判断，如果 useOffHeap 为 true，则会创建 OffheapIncrementalIndex，否则创建 OnheapIncrementalIndex。从名字上看就知道了，两者的不同之处是堆外内存的使用，OffheapIncrementalIndex 会使用堆外内存存储 Aggregator，采用全局的堆外内存池 OffheapBufferPool，它是 StupidPool 的子类，每次分配固定大小的堆外内存，其大小由参数 druid.processing.buffer.sizeBytes 设置，默认大小为 1GB。如果 Group By 查询的结果集很大，建议在查询上下文中设置 useOffHeap 为 true，减少 JVM GC 带来的性能影响。同时要增大 MaxDirectMemorySize，一般场景下：

$$\text{MaxDirectMemorySize} > (\text{processing.numThreads} + 1) \times \text{processing.buffer.sizeBytes}$$

这个公式是为了保障为每个处理线程能配置 sizeBytes 大小的内存。但并不包括 OffheapIncrementalIndex 的堆外内存的使用。

第 38 行中的 MaxResults 参数是指 IncrementalIndex 的最大容量，如果 Group By 查询返回结果集的容量大于该值，则会抛出 IndexSizeExceededException 异常。该参数由 druid.query.groupBy.maxResults 设置，默认值为 500000。这是系统的配置加载以后不能改变，为了灵活处理，可以在查询上下文中进行配置，然后和系统配置相比取最小值。

第55~77行是创建 Accumulator，在调用 Sequence 的 accumulate 方法时，迭代遍历内部的迭代器回调 Accumulator 的 accumulate 方法。它在类中传入的第一个参数是 IncrementalIndex，第二个参数是 row，代表每一条记录。accumulate 方法的逻辑很简单，就是将 Row 加入到 IncrementalIndex 中。IncrementalIndex 是线程安全的，在这个场景下会有多个线程执行 QueryRunner 返回 Sequence 并发添加到 IncrementalIndex 中。

6. CachingClusteredClient

CachingClusteredClient 是 Broker 中最核心的一个类，它按照查询的 interval 拆分成对 Segment 的查询，如果 Segment 在 Broker 中已有缓存则调用缓存的，没有则分发请求给历史节点或者实时节点（或是索引服务中的节点），然后合并结果。其执行流程如图 8-12 所示。

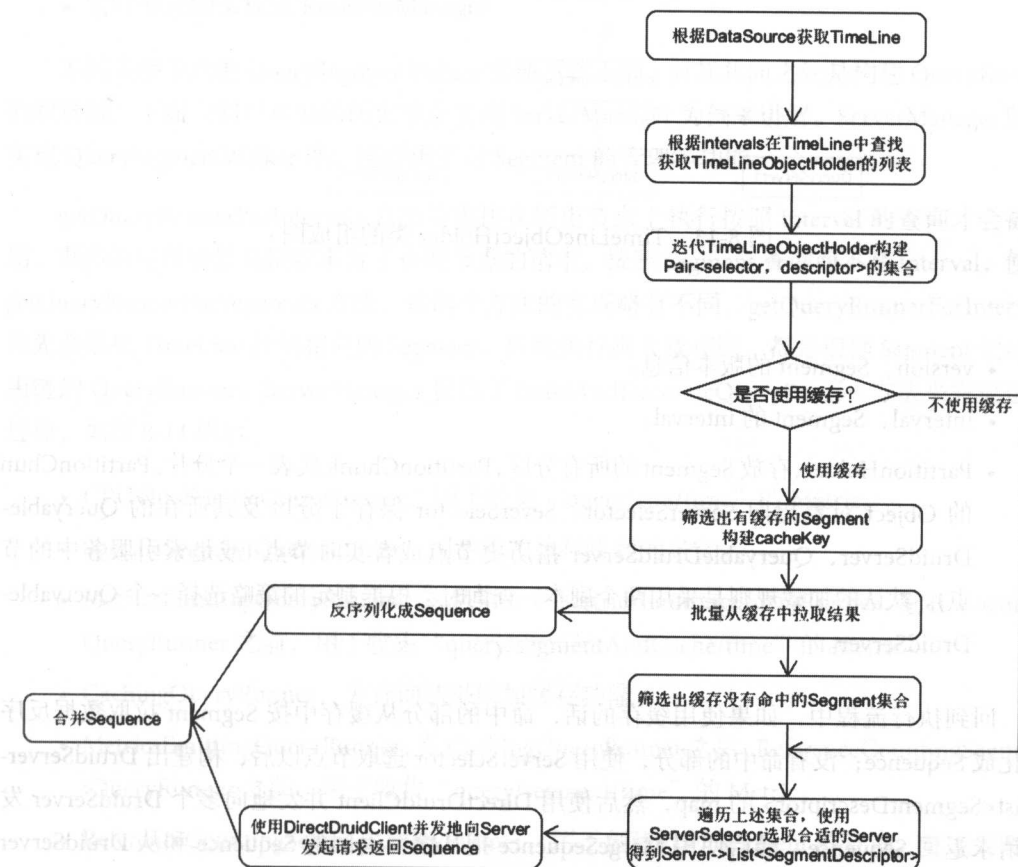


图 8-12 CachingClusteredClient 执行流程图

先来了解两个重要的类：TimeLine 和 TimeLineObjectHolder。TimeLine 由多个 TimeLineObjectHolder 组成，一个 TimeLineObjectHolder 对应于一个 Segment。该类的组成如图 8-13 所示。

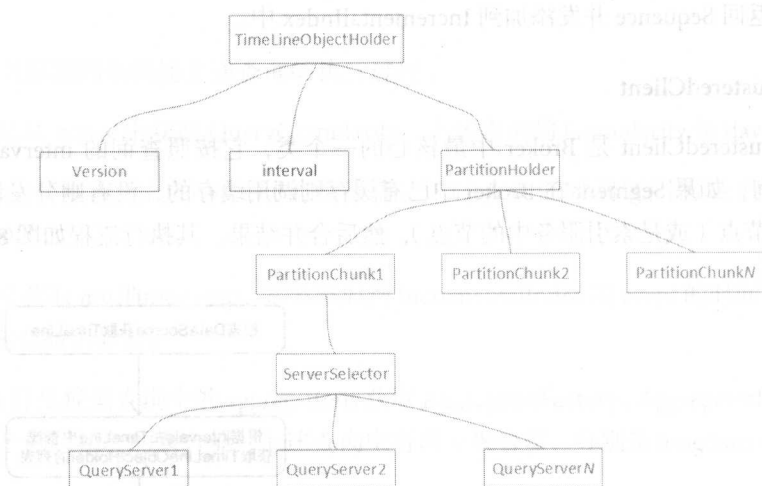


图 8-13 TimeLineObjectHolder 类的组成图

- version, Segment 的版本信息。
- interval, Segment 的 interval。
- PartitionHolder, 存放 Segment 的所有分区, PartitionChunk 代表一个分片。PartitionChunk 的 Object 对象指向 ServerSelector。ServerSelector 保存了分区及其所在的 QueryableDruidServer, QueryableDruidServer 指历史节点或者实时节点 (或是索引服务中的节点)。默认的加载规则是采用两个副本, 查询时会根据制定的策略选择一个 QueryableDruidServer。

回到执行流程中, 如果使用缓存的话, 命中的部分从缓存中按 Segment 拉取数据反序列化成 Sequence; 没有命中的部分, 使用 ServerSelector 选取节点以后, 构建出 DruidServer->List<SegmentDescriptor> 的 map, 然后使用 DirectDruidClient 并发地向多个 DruidServer 发起请求返回 Sequence, 最后使用 MergeSequence 把从缓存返回的 Sequence 和从 DruidServer 返回的 Sequence 合并, 这是 Scatter/Gather 模式的实现。值得一提的是, DirectDruidClient 采用的是基于 Netty 开发的 HttpClient, 通过异步 NIO 的方式, 并发地向多个 DruidServer 发起请求, 而不是通过采用多线程实现并发, 减少线程切换带来的开销。

7. QuerySegmentWalker

QuerySegmentWalker 有两个功能。

- 针对传入的多个 interval 构建 QueryRunner。
- 针对传入的多个 Segment 构建 QueryRunner。

每种类型节点的 QuerySegmentWalker 实现不同。

- 历史节点的实现是 ServerManager。
- 查询节点的实现是 ClientQuerySegmentWalker。
- 统治节点的实现是 ThreadPoolTaskRunner。
- 实时节点的实现是 RealtimeManager。

不同类型节点的 QuerySegmentWalker 实现逻辑不同，但其共同之处是构建 QueryRunner 的职责链。下面我们以典型的历史节点实现 ServerManager 为例来讲解。ServerManager 除了实现 QuerySegmentWalker 外，还提供了对 Segment 的管理等功能。

getQueryRunnerForIntervals 方法是直接在历史节点上执行按照 interval 的查询才会被调用，更多的应用场景是接收来自于查询节点的请求，按照 Segment 查询而不是 interval，使用 getQueryRunnerForSegments 方法。这两个方法的实现略有不同，getQueryRunnerForIntervals 首先会根据 TimeLine 找到相应的 Segment，后续执行则大致相同，都是根据 Segment 创建调用链的 QueryRunner。ServerManager 提供了 buildAndDecorateQueryRunner 方法来完成这个过程，如图 8-14 所示。

- CPUTimeMetricQueryRunner，用于收集“query/cpu/time”的 Metric。
- SpecificSegmentQueryRunner，用于修改执行线程的名字。
- MetricsEmittingQueryRunner，这是第一个 MetricsEmittingQueryRunner，它在 CachingQueryRunner 之前，用于收集“query/segmentAndCache/time”的 Metric。
- CachingQueryRunner，为查询结果添加缓存功能。
- MetricsEmittingQueryRunner，在 CachingQueryRunner 之后，ReferenceCountingSegmentQueryRunner 之前，用于收集“query/segment/time”的 Metric。
- ReferenceCountingSegmentQueryRunner，用于创建针对 Segment 的查询，它主要是调用 QueryRunnerFactory 的 createRunner 方法，创建具体查询类型的 QueryRunner，例如 Group By 查询则会创建一个 GroupByQueryRunner，在这个类中调用相应的引擎类执行查询。

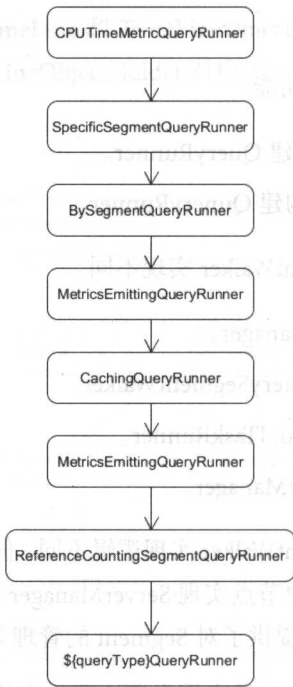


图 8-14 查询过程中的 Runner 步骤

8. QueryToolChest

QueryToolChest 类似于工具箱，用于辅助构建 QueryRunner 职责链。每一种查询类型都有自己的 QueryToolChest 实现。

QueryToolChest 提供了如下方法。

```
public abstract QueryRunner<ResultType> mergeResults(QueryRunner<ResultType> runner);
```

构建一个新的 QueryRunner，用于合并多个 QueryRunner 的结果集。

```
public abstract ServiceMetricEvent.Builder makeMetricBuilder(QueryType query);
```

用于构建 Metrics 中的查询相关信息，例如 numDimensions、numMetrics 和 numComplexMetrics 等信息。

```
public abstract Function<ResultType, ResultType> makePreComputeManipulatorFn(
    QueryType query,
    MetricManipulationFn fn
);
```

创建一个 Function 用于将一种 ResultType 转换成另一种 ResultType。它的经典应用场景是如果查询中含有复杂对象 Metric，Broker 合并从历史节点或者实时节点返回的结果时，使用该 Function 对复杂对象进行反序列化。MetricManipulationFn 提供了反序列化和将复杂对象类型转换为数值类型等常用 Function。

```
public Function<ResultType, ResultType> makePostComputeManipulatorFn(QueryType query,
    MetricManipulationFn fn)
{
    return makePreComputeManipulatorFn(query, fn);
}
```

其功能类似于 makePreComputeManipulatorFn，但是在合并结果之后使用。它的使用场景之一是把复杂对象 Metric 例如 HyperLogLog 转换成数值类型。

```
public abstract TypeReference<ResultType> getResultTypeReference();
```

使用 JSON 反序列化时获取对象的类型。

```
public <T> CacheStrategy<ResultType, T, QueryType> getCacheStrategy(QueryType query)
{
    return null;
}
```

获取缓存策略，用于决定如何从缓存中加载数据，以及从缓存中删除数据。这个是可选的，如果返回 null 则不使用缓存。

```
public QueryRunner<ResultType> preMergeQueryDecoration(QueryRunner<ResultType> runner)
{
    return runner;
}
```

封装传入的 QueryRunner，这个 QueryRunner 必须在合并结果之前执行。事实上，这个方法的返回值是传递给 mergeResults 方法的。它用在 Broker 层面，将传入的 QueryRunner 封装成 IntervalChunkingQueryRunner，将长时间区间的查询拆分成小段时间区间的查询，减轻对资源使用的影响。在查询上下文中设置 chunkPeriod，如果为 0 则不拆分。举个例子，如果有一个查询要查时间区间为 1 年的数据，当设置 chunkPeriod 为 P1M 时，则会把这个查询拆分成 12 个小查询并发执行。但要确保在 Broker 中设置 “druid.processing.numThreads” 为合适的值。

```
public QueryRunner<ResultType> postMergeQueryDecoration(QueryRunner<ResultType> runner)
{
    return runner;
}
```

封装传入的 QueryRunner，这个 QueryRunner 必须在合并结果之后执行。

GroupbyQueryToolChest 是辅助 Group By 查询的具体实现，是所有查询类型中最复杂的。我们以最重要的 mergeResults 方法为例，其整体执行流程如图 8-15 所示。

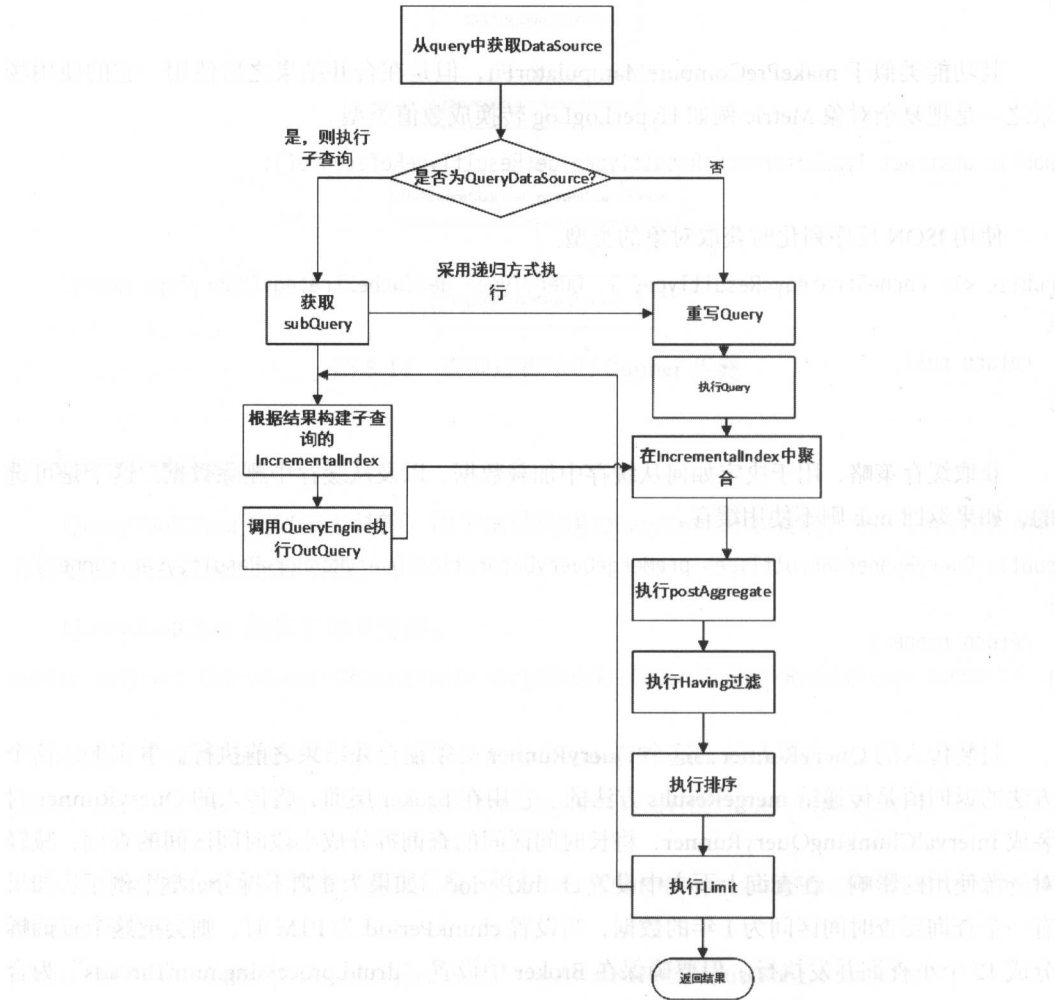


图 8-15 GroupbyQueryToolChest 执行流程图

Group By 查询支持子查询，也就是 QueryDataSource，所以首先判断 Query 中的 DataSource 是否为 QueryDataSource。子查询也必须是 Group By 查询，子查询的执行流程和普通查询是一致的。

- 重写 Query，去掉 Limit、Having 及 postAgg。
- 把从多个节点返回的结果在 IncrementalIndex 中聚合。
- 执行 PostAggregate，例如 JavaScript PostAggregate 使用 JavaScript 的 Function 完成较复杂的运算。这是可选项，当 Query 中包含 PostAggregatorSpecs 时才执行。
- 执行 Having 过滤，采用 FilterSequence 完成 Having 过滤。这也是可选项。
- 执行排序，将 Having 过滤以后的 Sequence 进行排序。
- 执行 Limit，按照 Limit 截取指定的条数。

再回到 QueryDataSource 中，此时子查询执行完成，根据子查询执行的结果构建 IncrementalIndex，然后使用 GroupByQueryEngine 在 IncrementalIndex 中执行外层 Query，其返回的 Sequence 继续执行 PostAggregate、Having、排序及 Limit 等流程，最后将结果返回。

TimeseriesQueryQueryToolChest 是辅助 Timeseries 查询的具体实现。还是来看 mergeResults 方法，在 TimeseriesQueryQueryToolChest 中设计了 ResultMergeQueryRunner 用于合并结果，除了 Group By 查询，几乎其他所有查询都是采用该类执行合并结果的。它利用 CombiningSequence 合并多个 Sequence，combine 不同于 concat，它会使用合并方法，把不同的记录合并在一起，例如 Timeseries 查询是把含有相同“时间列”的记录合并成一条，每种不同类型的查询都需要指定的排序和合并方法。ResultMergeQueryRunner 是一个抽象类，提供了两个抽象方法用于辅助构建 CombiningSequence。

```
protected abstract Ordering<T> makeOrdering(Query<T> query);  
protected abstract BinaryFn<T,T,T> createMergeFn(Query<T> query);
```

- makeOrdering，其作用是创建一个带有比较功能的 Ordering 对象，例如 Timeseries 查询的结果都是 <timestamp,metrics>，首先会比较 timestamp，如相同才会执行合并。
- createMergeFn，创建执行合并逻辑的函数，在 Timeseries 查询中，利用 AggregatorFactory 的 combine 方法合并两个 metric。

8.6.4 查询引擎

在学习查询引擎之前必须要熟悉 Cursor。先来看一下 Cursor 接口，它继承了 ColumnSelectorFactory，ColumnSelectorFactory 是一个工厂接口，提供了如下方法。


```
public DimensionSelector makeDimensionSelector(DimensionSpec dimensionSpec);  
public FloatColumnSelector makeFloatColumnSelector(String columnName);  
public LongColumnSelector makeLongColumnSelector(String columnName);  
public ObjectColumnSelector makeObjectColumnSelector(String columnName);
```

- **makeDimensionSelector** 方法，构建 **DimensionSelector**，用于获取 **Cursor** 指向的当前行的维度值。
- **makeFloatColumnSelector** 方法，构建 **FloatColumnSelector** 用于获取 **Cursor** 指向的当前行的 **Metric** 的值，并将其转换为 **Float** 类型。
- **makeLongColumnSelector**，其功能同 **makeFloatColumnSelector**。
- **makeObjectColumnSelector**，构建 **ObjectColumnSelector**，一般用于获取复杂对象的值，例如 **HyperUnique**。

Cursor 本身提供了一些和迭代遍历相关的方法。

```
public DateTime getTime();  
public void advance();  
public void advanceTo(int offset);  
public boolean isDone();  
public void reset();
```

- **getTime** 方法，获取 **Cursor** 指向当前行的时间。
- **advance** 方法，类似于迭代器的 **next** 方法，指向下一行。
- **advanceTo** 方法，跳到指定偏移的行。
- **isDone** 方法，判断游标是否结束，类似于迭代器的 **hasNext** 方法。
- **reset** 方法，重置游标到初始偏移处。

再来看一下 **DimensionSelector**，这也是非常重要的接口，用于辅助构建 **Cursor**。

```
public IndexedInts getRow();  
public int getValueCardinality();  
public String lookupName(int id);  
public int lookupId(String name);
```

- **getRow** 方法返回当前行的维度值，返回的 **IndexInts** 是 **int** 类型的列表，列表中元素是维度在字典编码中的值，采用列表是为了兼容处理多值维度。
- **getValueCardinality** 方法返回维度的基数，基数是指集合中不重复值的数量。例如以下

4行数据:

A

A,B

B

A,B

基数是 2, 字典编码以后是:

[0]

[0,1]

[1]

[0,1]

当 Cursor 指向第一行时, `getRow` 的返回值是 [0]。

- `lookupName` 方法会根据传入的字典编码值在字典中查询得到真实的维度值。

例如延续上面的例子, `lookupName(1)` 的返回值是 B。

- `lookupId` 和 `lookupName` 相反, 根据 Name 得到字典编码值, 这个方法常用于在获取 Id 以后, 再根据 Id 查找倒排索引中的 Bitmap。

Druid 没有执行计划, 而是采用固定的查询模型。每种类型的查询引擎实现不同, 但执行模式是通用的。查询引擎的执行模式如下。

- 根据 Query 的 Filter、Interval 等条件调用 `StorageAdapter` 的 `makeCursors`, 如果 Filter 不为空, 则采用 Bitmap 得到 Offset, 从而根据 Offset 构建 Cursor, 性能非常高。
- 迭代遍历 Cursor, 执行不同查询的具体逻辑。

构建 Cursor 和迭代遍历 Cursor 是相同的, 不同的是具体的查询逻辑。`QueryRunnerHelper` 提供了 `makeCursorBasedQuery` 方法来实现上述执行模式, 不同查询提供 Function 来实现具体的查询逻辑。

Select 查询的引擎是 `SelectQueryEngine`, Select 查询的功能是获取 Roll-up 以后的原始数据。它的执行逻辑很简单, 拉取每一条数据的 Dimension 和 Metric, 接下来分析其执行流程。

```
1 final Map<String, DimensionSelector> dimSelectors = Maps.newHashMap();
2 for (DimensionSpec dim : dims) {
3     final DimensionSelector dimSelector = cursor.makeDimensionSelector(dim);
4     dimSelectors.put(dim.getOutputName(), dimSelector);
5 }
6
```

```
7 final Map<String, ObjectColumnSelector> metSelectors = Maps.newHashMap();
8 for (String metric : metrics) {
9     final ObjectColumnSelector metricSelector = cursor.makeObjectColumnSelector(metric);
10    metSelectors.put(metric, metricSelector);
11 }
12
13 final PagingOffset offset = query.getPagingOffset(segmentId);
14
15 cursor.advanceTo(offset.startDelta());
16
17 int lastOffset = offset.startOffset();
18 for (; !cursor.isDone() && offset.hasNext(); cursor.advance(), offset.next()) {
19     final Map<String, Object> theEvent = Maps.newLinkedHashMap();
20     theEvent.put(EventHolder.timestampKey, new DateTime(timestampColumnSelector.
21         get()));
22     for (Map.Entry<String, DimensionSelector> dimSelector : dimSelectors.entrySet()) {
23         final String dim = dimSelector.getKey();
24         final DimensionSelector selector = dimSelector.getValue();
25         if (selector == null) {
26             theEvent.put(dim, null);
27         } else {
28             final IndexedInts vals = selector.getRow();
29             if (vals.size() == 1) {
30                 final String dimVal = selector.lookupName(vals.get(0));
31                 theEvent.put(dim, dimVal);
32             } else {
33                 List<String> dimVals = Lists.newArrayList();
34                 for (int i = 0; i < vals.size(); ++i) { dimVals.add(selector.lookupName(vals.get(
35                     i)));
36                 }
37                 theEvent.put(dim, dimVals);
38             }
39         }
40     }
41     for (Map.Entry<String, ObjectColumnSelector> metSelector : metSelectors.entrySet()) {
42         final String metric = metSelector.getKey();
```

```
43 final ObjectColumnSelector selector = metSelector.getValue();
44
45 if (selector == null) {
46     theEvent.put(metric, null);
47 } else {
48     theEvent.put(metric, selector.get());
49 }
50 }
51
52 builder.addEntry(new EventHolder(segmentId, lastOffset = offset.current(), theEvent));
53 }
54
55 builder.finished(segmentId, lastOffset);
```

第2~5行，根据维度创建 `DimensionSelector` 放在 `dimSelectors` 中。

第7~11行，根据 `Metric` 创建 `ObjectColumnSelector` 放在 `metSelectors` 中。

第13行，根据 `Segment` 获取开始 `Offset`。需要在 `Query` 的 `pagingSpec` 中设置 `pagingIdentifiers`。

第15行，使用 `Cursor` 的 `advanceTo` 方法跳到开始偏移处。

第18~53行，遍历 `Cursor`，直到 `Cursor` 达到最后位置或者 `PageOffset` 达到设定的阈值。

第22~39行遍历 `dimSelectors` 获取维度的值，并按照 `dim->dimValues` 放到 `event` 中，`event` 代表一条事件记录。首先利用 `selector` 的 `getRow` 获取 `indexedInt`，这是字典编码的整数列表，如果 `indexedInt` 的 `size` 为1，则取单值，然后再利用 `selector` 的 `lookupName` 方法获取真实的维度值；如果 `size` 大于1，则说明是多值维度，使用 `List` 存放多值。

第41~50行，遍历 `metSelectors` 获取 `Metric` 的值，都是使用 `ObjectColumnSelector`。

第52行，此时已经得到一行记录中所需的维度和 `Metric` 的值，放入 `SelectResultValueBuilder` 中，`SelectResultValueBuilder` 采用队列存放 `Select` 查询的结果。

8.7 Coordinator 模块

`Coordinator` 是 `Druid` 的中心协调模块，用于解耦各个模块之间的直接联系，负责 `Segment` 的管理和分发，用于控制历史节点上的 `Segment` 装载和移除，并且保持 `Segment` 在各个历史节点上的负载均衡。

Druid Coordinator 采用定期运行任务的设计模式，它包含一些不同目的的任务。它并不直接和历史节点发生调用关系，而是将 Zookeeper 作为桥梁，将指令发送到 Zookeeper 上，历史节点获取 Zookeeper 上的指令来装载和移除 Segment，所以每个历史节点不需要看到整个集群的情况。

Coordinator 装载和移除 Segment 的依据来自于一系列规则，这些规则可以通过 Druid 的管理工具或者参数来配置。这些规则包括：

- 永久装载（LoadForever）
- 时间段装载（LoadByInterval）
- 最近时段装载（LoadByPeriod）

另外，也可以设置移除（Drop）规则，类似于装载规则。

Coordinator 的代码入口在 \$druid\server\src\main\java\io\druid\server\coordinator\DruidCoordinator.java。大部分核心代码在相同目录中也可以找到。

DruidCoordinator 引入了不少管理类，用于获取 Segment 与集群的信息和管理能力。例如，MetadataSegmentManager 和 MetadataRuleManager 这两个接口的实现类都是通过 SQL 查询方式，从 MySQL 服务器获取规则和 Segment 的信息的。DruidCoordinator 类图如图 8-16 所示。

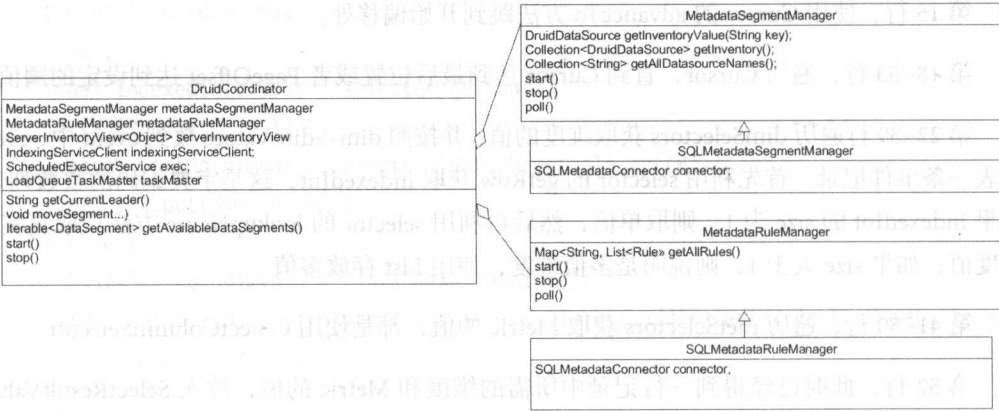


图 8-16 Druid Coordinator 类图

Coordinator 启动是从 start() 开始的，首先通过 Zookeeper 的 LeaderLatch 选取一个 Leader，确定好一个 Leader 后，这个 Leader 就会定期运行一些任务。

```
@LifecycleStart
public void start()
{
    ...
    createNewLeaderLatch();
    try {
        leaderLatch.get().start();
    } catch (Exception e) {
        throw Throwables.propagate(e);
    }
    ...
}

private LeaderLatch createNewLeaderLatch()
{
    final LeaderLatch newLeaderLatch = new LeaderLatch(...);

    newLeaderLatch.addListener(
        new LeaderLatchListener()
        {
            @Override
            public void isLeader()
            {
                DruidCoordinator.this.becomeLeader();
            }

            @Override
            public void notLeader()
            {
                DruidCoordinator.this.stopBeingLeader();
            }
        },
        Executors.singleThreaded("CoordinatorLeader-%s")
    );

    return leaderLatch.getAndSet(newLeaderLatch);
}
```

在 `becomeLeader()` 方法中, 由于已经确定当前节点为 `Leader`, 一些 `Runnable` 对象就可以依附当前的 `Coordinator`, 开始定期执行任务。

```
private void becomeLeader()
{
    synchronized (lock) {
        if (!started) {
            return;
        }

        try {
            leaderCounter++;
            leader = true;
            metadataSegmentManager.start();
            metadataRuleManager.start();
            serverInventoryView.start();
            serviceAnnouncer.announce(self);
            final int startingLeaderCounter = leaderCounter;

            final List<Pair<? extends CoordinatorRunnable, Duration>> coordinatorRunnables =
                Lists.newArrayList();

            coordinatorRunnables.add(
                Pair.of(
                    new CoordinatorHistoricalManagerRunnable(startingLeaderCounter),
                    config.getCoordinatorPeriod()
                )
            );

            if (indexingServiceClient != null) {
                coordinatorRunnables.add(
                    Pair.of(
                        new CoordinatorIndexingServiceRunnable(makeIndexingServiceHelpers(),
                            startingLeaderCounter
                        ),
                        config.getCoordinatorIndexingPeriod()
                    )
                );
            }
        }
    }
}
```

```

for (final Pair<? extends CoordinatorRunnable, Duration> coordinatorRunnable :
    coordinatorRunnables) {
    ScheduledExecutors.scheduleWithFixedDelay(
        exec,
        config.getCoordinatorStartDelay(),
        coordinatorRunnable.rhs,
        new Callable<ScheduledExecutors.Signal>()
        {
            private final CoordinatorRunnable theRunnable = coordinatorRunnable.lhs;

            @Override
            public ScheduledExecutors.Signal call()
            {
                if (leader && startingLeaderCounter == leaderCounter) {
                    theRunnable.run();
                }

                ...
            }
        }
    );
}
} catch (...) {
    ...
}
}

```

其中最重要的两个 Runnable 为 **CoordinatorHistoricalManagerRunnable** 和 **CoordinatorIndexingServiceRunnable**。

CoordinatorHistoricalManagerRunnable 包括多个具体的任务集合。

```

public CoordinatorHistoricalManagerRunnable(final int startingLeaderCounter)
{
    super(ImmutableList.of(
        new DruidCoordinatorSegmentInfoLoader(DruidCoordinator.this),
        new DruidCoordinatorRuleRunner(DruidCoordinator.this),
        new DruidCoordinatorCleanupUnneeded(DruidCoordinator.this),
        new DruidCoordinatorCleanupOvershadowed(DruidCoordinator.this),

```



```

new DruidCoordinatorBalancer(DruidCoordinator.this),
new DruidCoordinatorLogger(DruidCoordinator.this)
)
);
}

```

- **SegmentInfoLoader**: 装载 Segment 信息，删除无效的 Segment 信息。
- **RuleRunner**: 装载规则信息，并且应用到所有 Segment。
- **CleanupUnneeded**: 移除一些无效的 Segment、不在 MetaManager 中的 Segment。
- **Balancer**: 定时整理 Segment 分布的平衡性，移动部分 Segment 以平衡负载。

在 Druid 0.9 版本中，Balancer 的策略是 Interval based Cost，基本想法是定义任何两个 Segment 的 Cost Function，这个 Cost 是通过两个 Segment 被同时查询的可能性来定义的。如果两个 Segment 的时间靠近，它们就容易被同一个 Query 所覆盖到，因为查询的时间通常是连续的一段时间；如果两个 Segment 来自于同一个 DataSource，它们也容易被同一个 Query 所覆盖到。

Balancer 的想法就是尽量让那些容易被同一个查询覆盖的 Segment 分布在整个集群的不同历史节点上，最大利用集群的能力，避免大量查询集中在集群中的某些机器上。

Druid 0.9.0 之前的 Balancer 策略比较简单和粗暴。

```
final double cost = baseCost * recencyPenalty * dataSourcePenalty * gapPenalty;
```

- **recencyPenalty** 是考虑当前时间两个 Segment 的时间距离。
- **dataSourcePenalty** 是考虑数据源的同源性。
- **gapPenalty** 是考虑两个 Segment 的时间相邻性或者重叠。

Druid 0.9.1 引入了更加高效的机制，算法也更为复杂，有兴趣的读者可以访问 <https://github.com/druid-io/druid/pull/2972>，可以直接查看 \$druid/server/coordinator/CostBalancerStrategy.java 中的代码。

这一改进最大的特点是 Cost Function 中引入了可累加性 (Additive)，例如有 3 个 Segment，时间段分别为 A、B 和 C，那么 $\text{Cost}(A, B \text{ union } C) = \text{Cost}(A, B) + \text{Cost}(A, C)$ 。

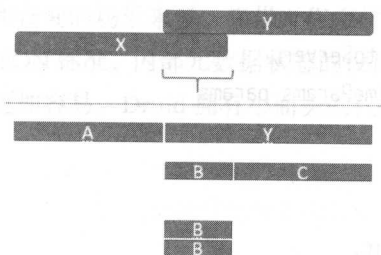
具体的 Cost 计算公式为：两个 Segment X 和 Y 分别覆盖时间段 $[x_0, x_1]$ 和 $[y_0, y_1]$ 。X 和 Y 的 Cost 函数如图 8-17 所示。

$$\text{Cost}(X, Y) = \int_{x_0}^{x_1} \int_{y_0}^{y_1} e^{-\lambda|x-y|} dx dy$$

图 8-17 新 Cost 公式

在实际计算中,有两种情况:一是两个 Segment 之间有时间段交集,例如, $x_0 < y_0 < x_1$; 二是两个 Segment 之间没有任何交集。

在计算第一种情况时,假设交集的点为 Y_0 ,那么 $\text{Cost}(X, Y) = \text{Cost}(A, Y) + \text{Cost}(B, C) + \text{Cost}(B, B)$, 如图 8-18 所示。



$$\text{Cost}(X, Y) = \text{Cost}(A, Y) + \text{Cost}(B, C) + \text{Cost}(B, B)$$

图 8-18 新算法的成本计算示意图

移动部分 Segment 的逻辑代码如下:

```
//DruidCoordinator.java
new DruidCoordinatorBalancer(DruidCoordinator.this),
...

//DruidCoordinatorBalancer.java
public DruidCoordinatorRuntimeParams run(DruidCoordinatorRuntimeParams params)
{
    ...
    for (int iter = 0; iter < maxSegmentsToMove; iter++) {
        final BalancerSegmentHolder segmentToMove = strategy.pickSegmentToMove(
            serverHolderList);

        if (segmentToMove != null && params.getAvailableSegments().contains(segmentToMove.
            getSegment())) {
            final ServerHolder holder = strategy.findNewSegmentHomeBalancer(segmentToMove.
                getSegment(), serverHolderList);
```

```

        if (holder != null) {
            moveSegment(segmentToMove, holder.getServer(), params);
        }
    }
}

...

protected void moveSegment(
    final BalancerSegmentHolder segment,
    final ImmutableDruidServer toServer,
    final DruidCoordinatorRuntimeParams params
) {
    ...
    loadPeon.loadSegment(segment,
        new LoadPeonCallback()
        {
            @Override
            public void execute()
            {
                try {
                    if (curator.checkExists().forPath(toServedSegPath) != null && curator.
                        checkExists().forPath(toLoadQueueSegPath) == null && !dropPeon.
                        getSegmentsToDrop().contains(segment)) {
                        dropPeon.dropSegment(segment, callback);
                    } else if (callback != null) {
                        callback.execute();
                    }
                } catch (Exception e) {
                    throw Throwables.propagate(e);
                }
            }
        }
    );
    ...
}

```

上面代码片段解释了 Segment 移动的管理过程, 作为定期运行的 DruidCoordinatorBalancer 任务, 通过 BalancerStrategy 对象获取最需要移动的若干 Segment, 而后通过 curator 操作 Zookeeper, 把 Segment 的来源和目标机器都写入 Zookeeper, 等待历史节点进行实际操作, 历史节点操作完成后将会删除该 ZK 节点。

8.8 小结

Druid 的代码整体实现非常简洁, 很少看到过多的冗余代码, 也很少使用复杂的设计模式。Druid 整体设计也充分考虑到扩展性, 在数据索引和查询部分都支持一定程度的扩展, 这对于一些有复杂要求的数据和查询的场景来说, 提供了很多扩展机会。另外, 由于 Druid 整体设计比较开放, 包括对外 JSON 标准, 内部元数据状态的设计也很清楚, 因此, 开发 Druid 周边工具和生态软件也变得更加容易。Druid 拥有小而美的代码库, 非常适合对数据库设计感兴趣的同学阅读和学习。

第9章

监控和安全

一个应用系统的成功不仅仰仗该系统所能提供的功能性需求，还取决于该系统是否能够被有效监控，以及建立起完善的安全保护机制。因为如果一个系统运行在不可知甚至不可控状态下，那么该系统不仅很难充分发挥出其效能，而且会具有较高的运维风险。基于这个道理，对于想成功实践 Druid 技术的用户来说，掌握监控 Druid 的必要技巧和了解其相关安全知识也就成为了一门必修课。本章会从 Druid 系统的监控、告警和安全等方面分享一些关于 Druid 运维管理相关的知识。

9.1 Druid 监控

为了精细化、实时地管理 Druid 系统，首当其冲的任务便是对 Druid 系统实施有效监控 (Monitoring)，及时并全面地获取其主要运行指标 (Metric) 的实时数值，然后通过这些指标对系统状态做出数据化的衡量，对系统表象进行根因分析，并进一步对系统做出必要的调整。幸运的是，正如其他成熟的项目一样，Druid 自身提供了丰富的监控指标与指标输出方法，因此我们可以轻松地利用这些特性对 Druid 进行有效监控。

9.1.1 Druid 监控指标

无论什么系统，如果想要被有效地监控，首要条件往往是其能够提供必要且充分的监控指标。Druid 便提供了丰富的指标供系统监控使用，并且提供了直接写日志和通过 HTTP 往外发送两种方式。不过，Druid 系统默认不会主动发送 Metric 信息，因此用户需要通过更改配置中“druid.emitter”参数的值来设置发送开关。

- noop: 默认值, 不往外发送任何 Metric 信息。
- logging: 往日志里面写 Metric 信息。
- http: 直接通过 HTTP 往外发送 Metric 信息。

无论使用哪种方式往外发送 Metric 信息, 所有的 Metric 信息都是 JSON 格式的数据, 并且拥有一些通用的字段。

- timestamp: Metric 生成的时间点的时间戳。
- metric: Metric 的名字。
- service: 发送 Metric 的服务名字。
- host: 发送 Metric 的主机名。
- value: Metric 的值。

一般来说, Druid 的监控指标可分为查询相关、数据消费相关与协调相关等种类。接下来, 我们将依次介绍一些主要的监控指标 (注: 主要基于 Druid 0.9.1.1 版本)。

1. 查询相关监控指标

查询节点的指标如下。

指标名称	描述	常规值
query/time	执行完一条 query 所花费的时间。单位为毫秒	< 1 秒
query/bytes	执行完一条 query 后所返回的数据量。单位为字节	
query/node/time	在独立的历史节点/实时节点上执行完一条 query 所花费的时间。单位为毫秒	< 1 秒
query/node/bytes	在独立的历史节点/实时节点上执行完一条 query 后所返回的数据量。单位为字节	
query/node/ttfb	查询节点首次从独立的历史节点/实时节点上收到查询返回的字节所花费的时间。单位为毫秒	< 1 秒
query/intervalChunk/time	查询一个间隔块所花费的时间。单位为毫秒。仅当间隔块被启用时本指标才有效	< 1 秒

历史节点的指标如下。

指标名称	描述	常规值
query/time	执行完一条 query 所花费的时间。单位为毫秒	< 1 秒
query/segment/time	查询一个 Segment 所花费的时间，其中包含将 Segment 从磁盘读取到内存页所花费的时间。单位为毫秒	几百毫秒
query/wait/time	等待一个 Segment 被浏览所花费的时间。单位为毫秒	几百毫秒
segment/scan/pending	等待被浏览的 Segment 的数量	接近 0
query/segmentAnd-Cache/time	查询一个 Segment 或者在 Cache 中命中数据（如果 Cache 在历史节点上被启用）所花费的时间。单位为毫秒	几百毫秒
query/cpu/time	执行完一条 query 所花费的 CPU 时间。单位为微秒	各有差异

实时节点的指标如下。

指标名称	描述	常规值
query/time	执行完一条 query 所花费的时间。单位为毫秒	< 1 秒
query/segment/time	查询一个 Segment 所花费的时间，其中包含将 Segment 从磁盘读取到内存页所花费的时间。单位为毫秒	几百毫秒
segment/scan/pending	等待被浏览的 Segment 的数量	接近 0

Cache 的指标如下。

指标名称	描述	常规值
query/cache/delta/*	自从上次发送后的 Cache 指标	
query/cache/total/*	总的 Cache 指标	
*/numEntries	Cache 的条目数量	各有差异
*/sizeBytes	Cache 的条目字节大小	各有差异
*/hits	Cache 的命中数	各有差异
*/misses	Cache 的没命中数	各有差异
*/evictions	Cache 的收回数量	各有差异
*/hitRate	Cache 的命中率	约为 40%

续表

指标名称	描述	常规值
*/averageByte	Cache 的条目平均字节大小	各有差异
*/timeouts	Cache 的超时数量	0
*/errors	Cache 的错误数量	0

以下是 Memcached 的相关指标，它们是从 Memcached 的客户端直接返回的（仅当配置中设置了“druid.cache.type=memcached”时），而不是从 Memcached 的服务端返回。

指标名称	描述	常规值
query/cache/memcached/total	Memcached 中 Cache 指标的数量	N/A
query/cache/memcached/delta	自从上次指标发送后，Memcached 中 Cache 指标的数量增量	N/A

2. 数据消费相关监控指标

仅当 RealtimeMetricsMonitor 被包含在实时节点的监控列表中时，Druid 才会发送出如下指标。与此同时，下面的指标均表示每次指标发送周期内所发生的增量值。

指标名称	描述	常规值
ingest/events/thrownAway	由于时间戳超出设定的 windowPeriod，被实时节点所丢弃的 event 数量	0
ingest/events/unparseable	由于不可以被成功解析，被实时节点所丢弃的 event 数量	0
ingest/events/processed	在每一个指标发送周期内，被实时节点所成功消费的 event 数量	等于在每一个指标发送周期内数据源所发送的所有 event 数量
ingest/rows/output	Druid 集群最后实例化的数据行数	基于所有的 event 数据聚合而成
ingest/persists/count	Druid 集群实例化数据的次数	由配置决定
ingest/persists/time	Druid 集群实例化数据所花费的时间。单位为毫秒	由配置决定，一般最多不超过几分钟
ingest/persists/cpu	Druid 集群实例化数据所花费的 CPU 时间。单位为纳秒	由配置决定，一般最多不超过几分钟

续表

指标名称	描述	常规值
ingest/persists/backPressure	Druid 集群实例化操作在排队等待的数量	0
ingest/persists/failed	Druid 集群实例化失败的次数	0
ingest/handoff/failed	文件传送操作失败的次数	0
ingest/merge/time	合并 Segment 碎片所花费的时间。单位为毫秒	由配置决定，一般最多不超过几分钟
ingest/merge/cpu	合并 Segment 碎片所花费的 CPU 时间。单位为纳秒	由配置决定，一般最多不超过几分钟
ingest/handoff/count	文件传送操作发生的次数	各有差异。如果系统运行正常，一般在每一个 Segment 完整粒度周期内其值大于 0



注意：如果 JVM 并不支持为当前线程记录 CPU 执行时间，那么 ingest/merge/cpu 与 ingest/persists/cpu 指标的值都将等于 0。

索引服务的相关指标如下。

指标名称	描述	常规值
task/run/time	执行任务所花费的时间。单位为毫秒	各有差异
segment/added/bytes	新创建的 Segment 的体积大小。单位为字节	各有差异
segment/moved/bytes	移动任务在执行时总共移动的 Segment 大小。单位为字节	各有差异
segment/nuked/bytes	清除任务在执行时总共删除的 Segment 大小。单位为字节	各有差异

3. 协调相关监控指标

以下指标均为 Druid 协调节点所设计，而且会在每次执行协调逻辑时被重置。

指标名称	描述	常规值
segment/assigned/count	被加载到 Druid 集群的 Segment 数量	各有差异
segment/moved/count	在 Druid 集群中被移动的 Segment 数量	各有差异

续表

指标名称	描述	常规值
segment/dropped/count	在 Druid 集群中由于过期而被删除的 Segment 数量	各有差异
segment/deleted/count	在 Druid 集群中由于规则设置而被删除的 Segment 数量	各有差异
segment/unneeded/count	在 Druid 集群中由于被设置为不再使用而被删除的 Segment 数量	各有差异
segment/cost/raw	执行任务所花费的时间。单位为毫秒	各有差异
segment/size	可被访问的 Segment 文件大小。单位为字节	各有差异
segment/count	可被访问的 Segment 文件数量	< max

4. 其他主要监控指标

接下来，再介绍一些同样比较重要的其他指标。

历史节点的指标如下。

指标名称	描述	常规值
segment/max	能被用于加载 Segment 数据文件的最大空间大小。单位为字节	各有差异
segment/used	目前已被加载的 Segment 数据文件的大小。单位为字节	< max
segment/usedPercent	目前已被加载的 Segment 数据文件的大小与最大可用空间的比值	< 100%
segment/count	目前已被加载的 Segment 数据文件的数量	各有差异

JVM 的指标：以下指标仅仅在 JVMMonitor 模块被加载时才会产生。

指标名称	描述	常规值
jvm/pool/committed	提交 pool	接近 max pool
jvm/pool/init	初始 pool	各有差异
jvm/pool/max	最大 pool	各有差异
jvm/pool/used	所使用的 pool	< max pool
jvm/bufferpool/count	Bufferpool 的数量	各有差异
jvm/bufferpool/used	Bufferpool 的使用量	接近容量大小

续表

指标名称	描述	常规值
jvm/bufferpool/capacity	Bufferpool 的容量大小	各有差异
jvm/mem/init	初始内存	各有差异
jvm/mem/max	最大内存	各有差异
jvm/mem/used	实际使用的内存大小	< 最大内存
jvm/mem/committed	实际提交的内存大小	接近最大内存
jvm/gc/count	垃圾回收的次数	< 100
jvm/gc/time	垃圾回收所花费的时间	< 1 秒

EventReceiverFirehose 的指标：以下指标仅仅在 EventReceiverFirehoseMonitor 模块被加载时才会被产生。

指标名称	描述	常规值
ingest/events/buffered	在 EventReceiverFirehose 的 Buffer 中排队的 event 数量	等于实际在 Buffer 中排队的 event 数量
ingest/bytes/received	EventReceiverFirehose 所接收的 event 的大小。单位为字节	各有差异

Sys 的指标：以下指标仅仅在 SysMonitor 模块被加载时才会被产生。

指标名称	描述	常规值
sys/swap/free	空闲的 swap	各有差异
sys/swap/max	最大的 swap	各有差异
sys/swap/pageIn	在 swap 中被调进的页	各有差异
sys/swap/pageOut	在 swap 中被调出的页	各有差异
sys/disk/write/count	写到磁盘的数量	各有差异
sys/disk/read/count	从磁盘中读的数量	各有差异
sys/disk/write/size	写到磁盘的数据量。单位为字节	各有差异
sys/disk/read/size	从磁盘中读取的数据量。单位为字节	各有差异

续表

指标名称	描述	常规值
sys/net/write/size	写到网络的数据量。单位为字节	各有差异
sys/net/read/size	从网络中读取的数据量。单位为字节	各有差异
sys/fs/used	文件系统的使用量。单位为字节	< max
sys/fs/max	文件系统的最大可使用量。单位为字节	各有差异
sys/mem/used	内存的使用量	< max
sys/mem/max	内存的最大可使用量	各有差异
sys/storage/used	磁盘空间的使用量	各有差异
sys/cpu	CPU 的使用量	各有差异

至此，我们对一些重要的 Druid 监控指标进行了介绍，接下来介绍如何使用这些监控指标对系统进行有效监控。

9.1.2 常用的监控方法

了解了监控指标的种类和意义后，接下来我们就需要采取具体的解决方案在实际的 Druid 系统环境中获取、存储、展示和分析这些指标数据，这样才能真正从这些指标数据中获益。上文介绍过 Druid 往外发送指标数据主要有两种方式，即发送到日志文件，以及通过 HTTP 往外发送，因此我们在设计具体的接收指标数据的方案时也可以从这两种方式切入，最后形成种类不同但最终作用一致的解决方案。本节将重点介绍笔者在实际工作中使用过的两种解决方案，抛砖引玉，希望大家能够得到一些启发。

1. 基于 HTTP 方法的监控系统设计方案

如果 Druid 集群采用 HTTP 方法往外直接发送监控指标，就需要有一个 HTTP Server 来接收并处理这些指标数据。在这种情况下，一个比较普遍的使用场景是该 HTTP Server 往往会将接收到的指标数据不加任何处理便直接转发到消息缓存系统（如 Kafka）中，然后再用单独的 Consumer 应用从消息缓存系统中消费数据，并存储到一个适合存储与分析指标数据的数据库中——Druid 本身便满足这个要求，而实际上很多 Druid 大用户（比如 MetaMarkets）在其生产系统中便使用了这种方法。此时，用户便可以通过直接分析对应于 Druid DataSource 中的指标数据，或者使用诸如 Druid Pivot 类似的 UI 工具来进行展示和交互式分析。

如果你的使用场景与上述场景类似，就可以选择自己编写一个能够满足功能要求的 HTTP Server。幸运的是，你不需要这样做，因为目前已经有了一个叫作 `druid-metrics-to-kafka` 的开源项目帮助你完成了同样的工作——直接接收并转发通过 HTTP 发送过来的指标数据到 Kafka 消息缓存系统中，所以你可以直接利用该项目来加速实现监控系统。关于开源项目 `druid-metrics-to-kafka` 的具体使用方法，我们将在第 11 章“Druid 生态与展望”中分析，本章不作过多描述。至此，我们便可以将该基于 HTTP 方法的方案简化并映射为一个具体的架构实现，如图 9-1 所示。

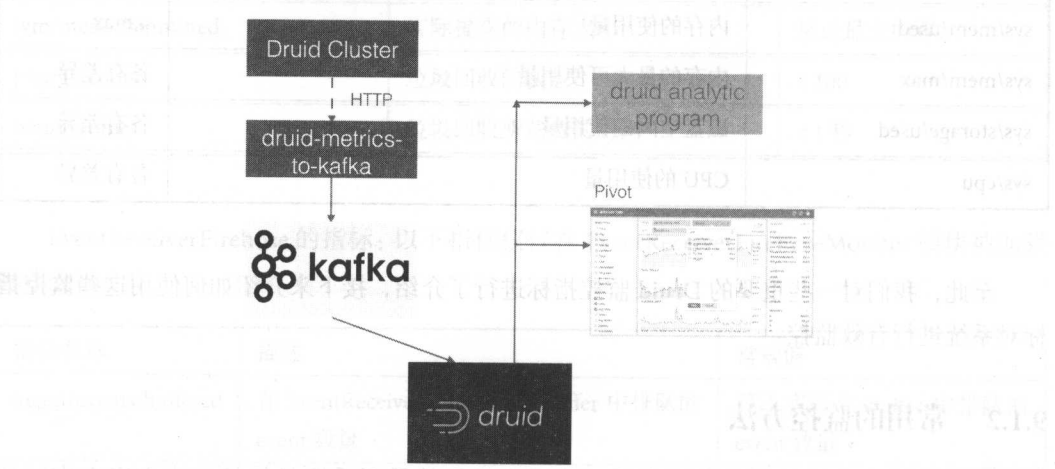


图 9-1 基于 HTTP 方法的监控系统架构示意图

对于 Druid 指标的可视化展示与交互式分析，除了使用开源的 Pivot 等工具外，也可以考虑使用其他的商用工具，如 OneAPM Cloud Insight 等，如图 9-12 所示。



图 9-2 OneAPM Cloud Insight 对 Druid 监控的支持

2. 基于日志方法的监控系统设计方案

如果 Druid 系统采取的是基于日志的方法往外直接发送监控指标,那么你首先需要有对应的程序能够获取不断更新的日志内容,从中解析并提取出所有感兴趣的 Druid 监控指标数据,然后将这些指标数据存储到适当的数据库中,接下来最好还有可视化工具帮助查看和分析所获得的指标数据。当然,有很多方法可以实现这些功能,比如编写能够读取新增日志内容的应用程序并在其中使用正则表达式来解析和提取出所期待的指标数据,也可以和上文所介绍的方案一样使用 Druid 本身来存储这些指标数据。接下来将介绍一个基于 Elasticsearch 技术栈构建的解决方案——该技术的确很适合目前的场景,帮助我们加速从零创建一个满足要求的监控系统。

Elasticsearch 是一个基于 Lucene 技术的全文检索平台,它提供了一个分布式且多租户的搜索引擎,目前可以说是业内使用最广泛的一款产品。除了拥有出色的搜索引擎这一优点外,Elasticsearch 还拥有较为完善的生态系统,即有很多其他项目围绕着 Elasticsearch 展开并提供了许多补充性功能,这使得 Elasticsearch 技术的可接受度得以大大提高——这也正是该技术得以普及的一个重要因素。Elasticsearch 技术栈中的 Elasticsearch、Logstash 和 Kibana 三个模块常常被一起使用,因此被广大用户简称为 ELK。实际上,ELK 经常被用户用来构建自己的日志分析与监控系统,而 Druid 正好可以将指标数据输出到日志文件中,所以我们也实际工作中利用相似的技术栈来构建自己的 Druid 监控系统并取得了很好的应用效果。因此,特在本章向读者朋友们做相应介绍。

首先介绍一下 ELK 中除 Elasticsearch 之外的两个项目。

- Logstash: 主要用来收集日志信息,并可以对其进行一些自定义操作,然后再将数据信息传送到指定的服务器上。简而言之,它适合在数据进入 Elasticsearch 之前为其做数据 ETL (抽取 Extract, 转换 Transform, 加载 Load) 操作。
- Kibana: 为 Elasticsearch 提供一个基于 Web 的可视化展示与分析接口。

虽然 Logstash 是一款轻量级且使用方便的 ETL 工具,但我们的解决方案中并没有使用它,而是使用了 Apache Flume,主要原因如下。

- 同样是主要用于 ETL 任务的 Apache Flume,作为 Apache 顶级项目,比 Logstash 有着更广泛的开源基础与社区活跃度,并且已经成为 Hadoop 生态圈的事实标准。
- 以 Hadoop 为中心的大数据技术生态圈基本以 Java 为主要开发语言,Flume 也一样。但是 Logstash 却不是用 Java 开发的,如果我们想对自己的以 Java 为主的大数据平台中的 ETL 工具进行二次开发或改进时,Logstash 的成本可能会高于 Flume。

同时，我们采用了 Kafka 作为消息缓存系统，因此 Druid 监控系统的数据流是：Log → Flume → Kafka → Flume → Elasticsearch → Kibana。系统架构如图 9-3 所示。

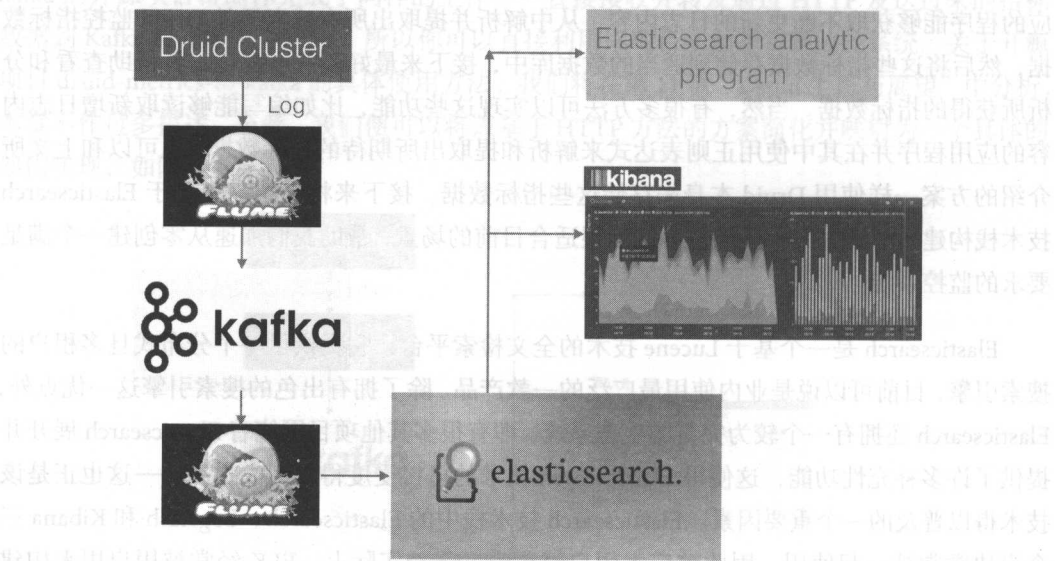


图 9-3 基于日志方法的监控系统架构示意图

下面展示几张我们在实际生产环境中基于上述 Elasticsearch 技术栈构建的 Druid 监控系统的效果图（见图 9-4、图 9-5 和图 9-6），希望能够给读者朋友们一些感性认识。

首先，我们看看通过监控系统来分析 Druid 集群的查询性能问题。根据前面章节的介绍，我们知道查询节点、实时节点和历史节点都能决定 Druid 集群的查询性能。也就是说，即使客户通过查询节点进行查询时感觉性能不好，也不能确定一定是查询节点的问题，因为也可能是实时节点和历史节点的问题。所以，如果用户想通过监控系统对查询性能问题进行剖析，就需要同时对这三个角色的节点进行统计分析方可得到较为客观的判定。因此，我们的监控系统收集了这三个角色上的所有 request time 指标值，然后通过 Kibana 端图像化展示便可以轻松地得知查询性能问题究竟出于何处。

我们也可以通过收集 events processed 指标来统计实时节点的成功消费数据情况。

当然，我们还可以通过收集 events thrown away 指标来统计实时节点对超时数据的丢弃情况。

我们就先展示这几个实际生产系统中的监控指标及其对应的监控界面，希望能够起到抛砖引玉的效果。

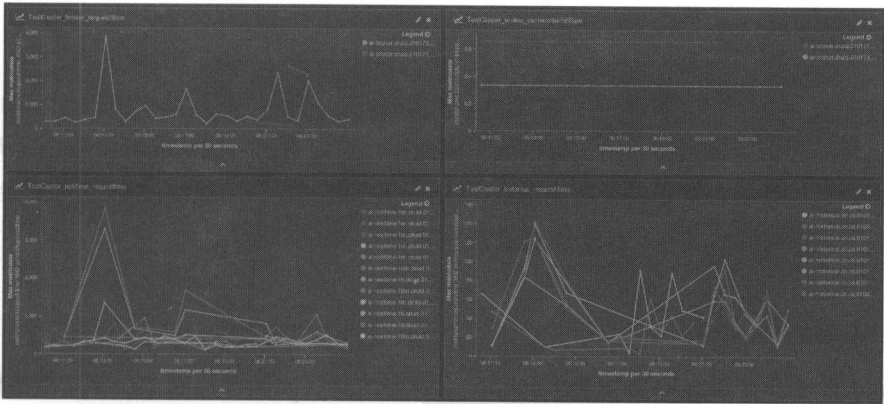


图 9-4 在监控系统中分析 Druid 查询性能问题

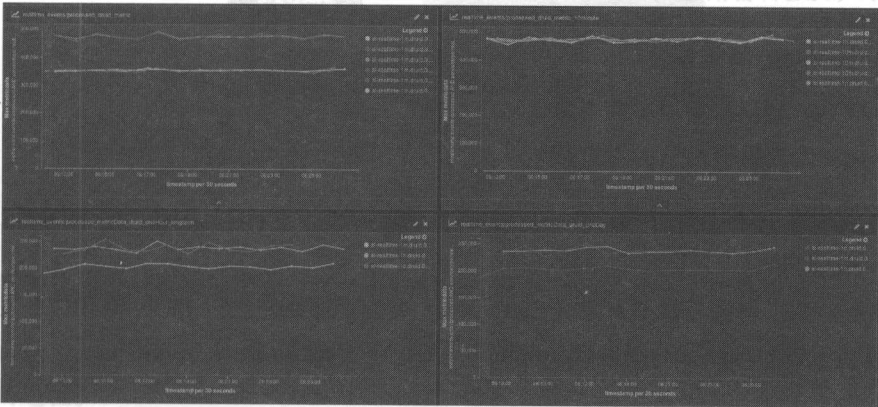


图 9-5 在监控系统中统计实时节点的成功消费数据情况

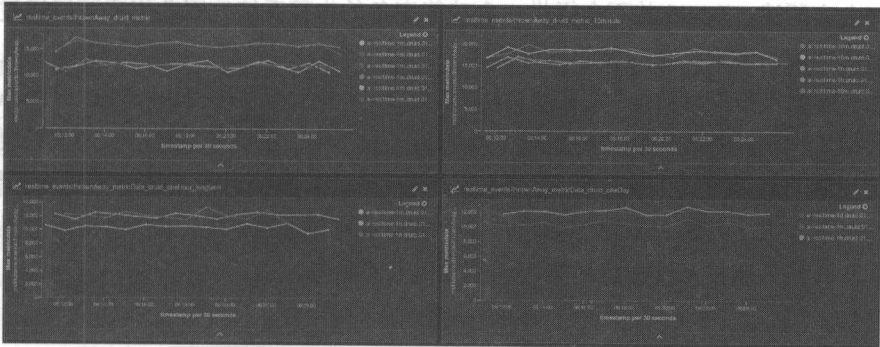


图 9-6 在监控系统中统计实时节点对超时数据的丢弃情况

9.2 Druid 告警

通过上文的介绍，大家可以了解到 Druid 主要通过提供指标数据来实现监控功能的支持，自己并不会直接提供完整的监控模块，因此用户需要自己搭建起能够接收并分析 Druid 指标数据的监控系统。对于告警功能来说，Druid 也采取了同样的措施——它并不会提供一个功能完备的告警系统，而仅仅是提供最基础的告警信息给用户。

9.2.1 Druid 告警信息

Druid 系统会在系统发生异常情况时产生告警信息。如同普通的 Druid 监控指标一样，其告警信息默认也是关闭不开启的，而且也是通过发送到日志文件，或者通过 HTTP 往外发送两种方式发送告警信息的。Druid 告警信息一般包含的字段如下。

- timestamp: 告警信息产生时间的时间戳。
- service: 发出告警信息的服务。
- host: 发出告警信息的主机。
- severity: 告警信息的级别。
- description: 告警信息的描述。
- data: 如果告警信息是异常 (Exception) 类型，那么它会以 JSON 格式往外发出，并且会包含 exceptionType、exceptionMessage 和 exceptionStackTrace 字段

9.2.2 Druid 与告警系统的集成

正如上文所述，Druid 并不会提供一个功能完备的告警系统，因此用户可以通过其他具备告警系统的模块实现一个完整的告警功能。比如，如同监控系统一样，可以使用 Logstash 或 Flume 来完成告警信息的抽取，并通过用户自己编写的指标发送模块 (Metrics Forwarder)，将指标数据发送到具备告警功能的系统 (Alert System) 中。在告警系统中，用户只要提前设定好告警规则与告警方式，在接收到 Druid 发送过来的告警信息时系统就可触发告警行为。在通常情况下，大多数解决方案都会同时包含监控系统与告警系统，图 9-7 所示便是告警系统与上文所介绍的监控系统集成的架构示例，供大家参考。

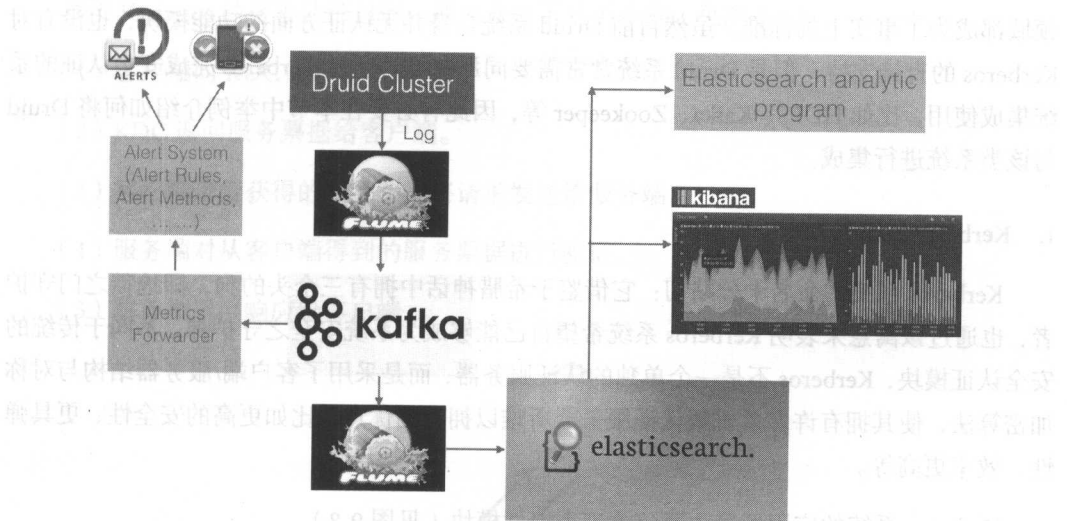


图 9-7 Druid 告警系统与监控系统集成架构示例

9.3 Druid 安全

如同对其他信息系统的考虑一样，我们对 Druid 安全的关注也主要分为两个部分。

- 认证（Authentication）：证明用户的确是他自己声明的那个用户的过程，即身份验证的过程。
- 授权（Authorization）：限定用户在系统中的权限范围，即指明某个用户在哪些系统模块中有哪些行为权限。

然而，遗憾的是，即使拿目前 Druid 的最新版本（0.9.1.1）来看，它自身无论在认证还是授权方面都做得很少，甚至可以说基本上没有任何实质性的相关功能。但是这并不表示我们在实际工作中使用 Druid 时就可以完全不考虑安全问题。相反，在很多企业级或大型互联网系统中，安全往往都是必须要考虑并完善的部分。因此在本节中，笔者就从如何将 Druid 系统与其他系统进行安全集成这个角度来介绍实际工作中的一些相关经验，希望能够帮助读者对基本的安全问题有所了解。

9.3.1 Druid 与利用 Kerberos 加强安全认证的系统集成

在当今的分布式技术领域，很多组件都会选择 Kerberos 来为其加强安全认证方面的功能，或者至少也将 Kerberos 当作其安全认证方面的组件选择之一，因此 Kerberos 几乎在很多

领域都成为了事实上的标准。虽然目前 Druid 系统自身并无认证方面的功能模块,也没有对 Kerberos 的直接支持,但是 Druid 系统常常需要同那些自身通过 Kerberos 完成安全认证的系统集成使用,比如 HDFS、Kafka、Zookeeper 等,因此有必要在本节中举例介绍如何将 Druid 与该类系统进行集成。

1. Kerberos 简介

Kerberos 系统的命名十分贴切:它借鉴于希腊神话中拥有三个头的狗,即地狱之门守护者,也通过该寓意来表明 Kerberos 系统希望自己能够成为系统安全之守护神。不同于传统的安全认证模块, Kerberos 不是一个单独的认证服务器,而是采用了客户端/服务器结构与对称加密算法,使其拥有许多单机版认证服务器所难以拥有的优点,比如更高的安全性、更具弹性、效率更高等。

Kerberos 系统的应用场景主要包含三个组件模块(见图 9-8)。

- 客户端 (Client): 发起服务请求的系统。
- 服务端 (Service): 真正提供服务的系统。
- 密钥分发中心 (Key Distribution Center, KDC): 第三方服务,用来对不同计算机的身份进行验证,并建立密钥以保证计算机间安全连接。KDC 又包含两个子模块——Kerberos 认证服务器和票据授权服务器。

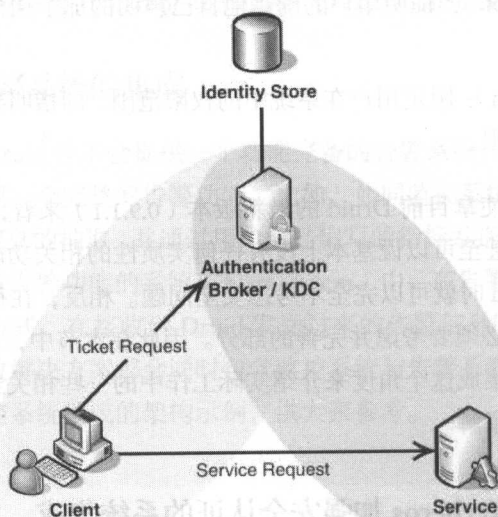


图 9-8 Kerberos 系统模块

基于 Kerberos 的完整认证过程一般包含以下几个步骤（见图 9-9）。

- （1）客户端向 KDC 请求认证票据。
- （2）KDC 返回服务票据给客户端。
- （3）客户端带着获得的服务票据将请求发送给服务端。
- （4）服务端对从客户端得到的服务票据进行验证。
- （5）服务端返回响应给客户端。

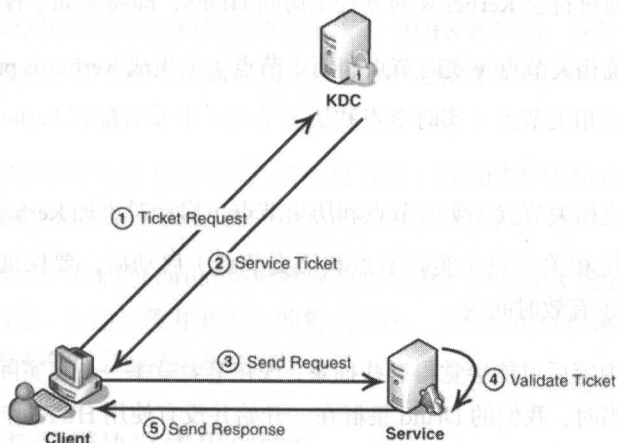


图 9-9 Kerberos 认证过程的步骤

关于 Kerberos 的知识介绍，无论在互联网上还是各种相关书籍中已有很多资料，而且这部分也不是本书的重点，因此就不在此处进行更多的描述了。

2. Druid 与使用了 Kerberos 的 HDFS 进行集成

现在很多系统的安全认证模块都基于 Kerberos 搭建而成，比如 HDFS、Kafka、Zookeeper 等，而它们都可能与 Druid 系统进行集成，但是本书没有办法穷举 Druid 与这些系统集成方法，因此仅仅拿与启用了 Kerberos 的 HDFS 系统集成作为示例，从而说明这一类集成的大概思路。

Hadoop 在其版本 1.0.0 之前存在着较大的安全问题，简单总结起来有以下两点。

- 用户与服务器之间的安全连接问题。NameNode、DataNode 和 JobTracker 都没有安全认证，因此任何用户登录都可以伪装成其他用户以执行几乎任何操作，从而导致较高的系统风险。

- 服务器之间的安全连接问题。DataNode 和 TaskTracker 没有安全认证，因此用户可以伪装成 DataNode 和 TaskTracker 接受来自 NameNode 和 JobTracker 的任务。

正因如此，Hadoop 在其版本 1.0.0 之后，引入了 Kerberos 模块作为其安全认证的基石，从而完美地解决了上述安全问题。所以，现在很多企业的 Hadoop 集群往往会启用基于 Kerberos 的安全认证功能模块。因此，在这些环境中如果要添加 Druid 系统，而且 Druid 系统要把 HDFS 当作其 DeepStorage，就一定要考虑 Kerberos 的认证问题。

简单总结来说，要想使 Druid 系统成功与启用了 Kerberos 安全认证模块的 HDFS 集成，就要将 Druid 系统通过符合 Kerberos 的方式来访问 HDFS，即需要如下操作。

- 在 Druid 系统相关节点（实时节点和历史节点）上生成 Kerberos principal 和 keytab。
- 在 Druid 系统相关节点（实时节点和历史节点）上安装配置 kerberos 客户端和 HDFS 客户端。
- 在 Druid 系统相关节点（实时节点和历史节点）启动时添加 Kerberos 所依赖的文件。
- 在 Druid 系统相关节点（实时节点和历史节点）启动后，要保证其所需的 Kerberos TGT 一直处于有效期内。

大家看了上述内容后可能会觉得有些抽象，现在笔者就举一个在实际工作中碰到的情况来进行实例说明。当时，我们的 Druid 集群在一开始并没有使用 HDFS 作为 DeepStorage，后来要将 DeepStorage 迁移到一个启用了 Kerberos 安全认证模块的 HDFS 集群上，所以执行了以下步骤。

（1）创建 Druid 节点所需的 Kerberos principal 和 keytab。

- 尽量为每一个需要访问 Kerberized HDFS 的 Druid 节点都分别创建一个 principal，principal 的命名格式为：[任意 principal 名称]/[所在主机的主机名]@[realm 的名称]（仅供参考）。其中 principal 的名称可以任意；所在主机的主机名可以通过 hostnamectl 命令查看当前机器的静态主机名，或者使用当前主机在 Druid 集群中的主机名；realm 的名称请参考 KDC 的配置，必须使用与当前 Kerberized HDFS 一致的 realm，比如 myname/x86-ubuntu@DRUID.COM。
- 生成 keytab（参考命令：xst-norandkey -k appd.keytab appd/x86-ubuntu HTTP/x86-ubuntu），复制到 Druid 节点的相应目录下，并保证 Druid 进程用户有读权限。

(2) 在所有 Druid 的实时节点和历史节点上安装配置 kerberos 客户端。

- 安装 kerberos 客户端。示例命令：`apt-get install krb5-clients (ubuntu)`。
- 检查 `/etc/krb5.conf` 配置文件。其中参数 `ticket_lifetime` 为 TGT 超时时长，`renew_lifetime` 为 TGT 刷新的生命周期时长。
- 添加 Crontab 任务，定期执行 `kinit` 命令来更新 TGT（供参考的 `kinit` 命令：`kinit -k -tappd.keytab appd/localhost@ONEAPM.COM`）。需要注意的是，`kinit` 的更新频率必须小于 TGT 的有效期，即必须在 TGT 失效前就更新 TGT。

(3) 在所有 Druid 的实时节点和历史节点上安装 HDFS 客户端，保证 Druid 进程用户能够访问其 HDFS 配置文件和 HDFS 所需要的 jar 包，并能够读写 HDFS 上分配给 Druid 集群用的目录。

(4) 在 Druid 系统相关节点（实时节点和历史节点）启动时添加 Kerberos 所依赖的配置文件（如 `-Djava.security.krb5.conf=/etc/krb5.conf`）和其他文件等。

执行了上述步骤后，我们的 Druid 集群就成功与启用了 Kerberos 安全认证模块的 HDFS 集群进行了集成。当然，上述步骤并非唯一的集成方法，大家也可以根据自己对 Kerberos 的了解情况以及实际需求来制定自己的集成方案。

9.3.2 集成外部权限模块完成用户授权

目前 Druid 并没有授权方面的具体功能，但是值得庆幸的是，Druid 开发社区已经在进行相关的开发任务（主要是对 `DataSource` 进行用户权限的划分与管理），我们期待社区能够早日完成该开发任务，以便广大 Druid 用户可以尽早将安全模块应用到自己的生产系统中。但是在此之前，我们依然需要考虑 Druid 的一些授权问题，否则可能会导致一些比较严重的问题。比如，假设对所有 Druid 用户都不设置权限的话，可能会导致如下问题。

- 任何用户都能够访问 Druid 集群的任何 `DataSource`，这可能会导致有些保密程度较高的数据也会被不加限制地访问到，从而造成泄密。
- 任何用户都可以查询任意时间范围内的数据，这可能会导致 Druid 集群的负载难以管控与分配，而某些超长时间范围的查询甚至有可能导致系统陷入不稳定状态，从而影响到其他正常查询的进行。
- 任何用户都能够向任何 `DataSource` 写入数据，这可能会导致由于误操作而带来的数据准确度变差问题，甚至给系统带来经济方面的损失。

正因如此，即使 Druid 自身不提供用户权限管理，我们也应该在 Druid 之外提供一些权限管理模块来补齐 Druid 在这方面的缺陷。举个具体的例子，我们可以让用户通过一个代理模块来访问 Druid 集群，并且同时将用户自己的系统以及权限管理模块与该代理模块进行集成，从而形成一套完整且易与其他已有系统进行集成的权限模块。图 9-10 所示是一个简单的示例。

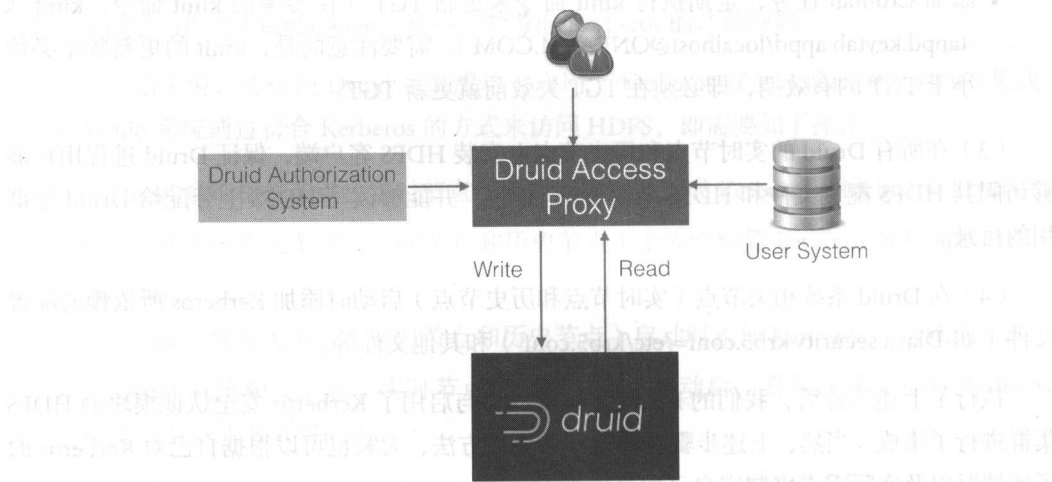


图 9-10 用户自定义权限管理模块与 Druid 系统的集成

9.4 小结

通过本章的介绍，相信读者朋友们已经对 Druid 的监控与安全相关知识有了不少了解。实践出真知，大家可以在使用 Druid 的过程中活学活用这些知识，这样就能帮助自己更好地驾驭 Druid，使它发挥出更大的能量。

第10章

实践和应用

自 Druid 诞生以来，它凭借着自己优秀的特质，不仅逐渐在技术圈收获了越来越多的知名度与口碑，并且陆续成为了很多技术团队解决方案中的关键一环，从而真正在很多公司的技术栈中赢得了一席之地。在成为 Druid 用户的公司中不乏很多国外赫赫有名的互联网企业（如雅虎、eBay、MetaMarkets 与 Hulu 等），也包含了很多国内同样声名遐迩的知名公司（如小米、优酷土豆、腾讯、蓝海讯通等），同时也覆盖了许多朝气蓬勃的创业团队。

Druid 收获了如此多的用户，那么大家一定都是不约而同地看中了 Druid 的某些关键属性，因此不同用户对 Druid 的使用也势必有许多相似之处，这可以说是件顺理成章的事情。然而，Druid 用户其实分布在许多不同的行业和领域，大家使用 Druid 的业务基础与技术背景也大相径庭，所以各自对 Druid 的使用场景也有许多迥异之处——而这些独到的地方也恰恰是有助于更加深入学习、多角度理解 Druid 的宝贵材料。因此，本章会介绍一些背景差异较大的公司对 Druid 的使用实践，希望能够帮助读者事半功倍地学习 Druid 这项年轻的技术。

10.1 小米

小米公司正式成立于 2010 年 4 月，是一家专注于高端智能手机、互联网电视以及智能家居生态链建设的创新型科技企业。

“让每个人都能享受科技的乐趣”是小米公司的愿景。小米公司应用互联网模式开发产品，用工匠精神做产品，用互联网模式节省了中间环节，致力于让全球每个人都能享用来自中国的优质科技产品。

除了手机之外，小米公司在互联网电视机、智能路由器和智能家居产品等领域也颠覆了传统市场。小米生态链建设以开放、合作共赢的策略，和业界合作伙伴一起推动智能生态链建设。

大数据作为小米公司的科技战略，不断地帮助小米产品提高用户体验和改进产品质量。2016年年中，小米 MIUI 设备总激活数超过 2 亿，月活用户超过 1.5 亿。数据处理和分析的任务变得非常艰巨和重要。

于是，小米云平台团队应运而生。小米大云平台承担着为公司提供海量数据的存储与处理服务平台的任务，整个平台是基于 Hadoop 生态系统的若干明星产品搭建的，包括数据采集、数据存储、数据管理、数据分析、机器学习和算法、可视化等部分。

小米大云平台整个技术架构和一些技术选型如图 10-1 所示。



图 10-1 小米大云平台技术架构示意图

Druid 在数据分析层帮助实时收集海量的事件数据，快速进行商业分析，在多个场景中都有应用。这里介绍 Druid 在小米统计产品和小米广告平台中的部分技术实践。

10.1.1 场景一：小米统计服务

小米统计是小米为 App 开发者提供的移动应用数据统计服务，帮助开发者通过数据了解应用发展状况、渠道推广效果和用户参与情况等信息，使开发者可以更好地优化体验和运营，促进产品不断发展进步。小米统计的入口是 tongji.xiaomi.com，服务界面如图 10-2 所示。



图 10-2 小米统计服务界面

实时的数据分析重要需求，在产品发展过程中，也经历了几个技术阶段，这几个阶段并非完全互斥，而是应用于不同的场景和时间。

第一阶段：数据存储存储在 Hadoop 中，通过 MapReduce 的脚本进行分析和处理。有一部分复杂的任务会以天为单位被执行，并且最后会将结果写入到如 MySQL 的 RDBMS 中。

第二阶段：在业务发展过程中 MySQL 很快变成了瓶颈，有两个原因，一是数据库的 Schema 更改成本高，新业务不断需要增加新列和新表，流程烦琐而且需要进行 Schema 设计；二是在进行大量写操作的情况下，数据库的负载增加会导致数据库的读性能下降，而且偶尔有死锁的现象。为了解决这些问题，引入了 HBase 作为主要存储数据库，利用 HBase 的列族，方便增加数据列。另外，HBase 的可用性也高于 MySQL。

第三阶段：为了改进数据的实时性，后期增加了 Storm 分布式计算模式，使用 Storm 可以方便地进行各种复杂的数据处理，各种聚合和处理需要通过程序实现，增加一个数据维度，改动比较大，需要从上游到下游整体修改。这种方式的优点是可靠性好，数据处理能力强，可以进行各种角度的优化。

第四阶段：小米统计的很多数据查询都是选择一些指标和过滤条件，很多场景类似于传统的数据仓库，因此引入 Druid 处理一些标准报告的实时数据查询场景。数据流会依次通过 Kafka 和 Tranquility，最后进入 Druid 集群。Druid 集群最终能提供最近一天的数据查询功能，并且允许用户直接访问。

小米统计数据流如图 10-3 所示。

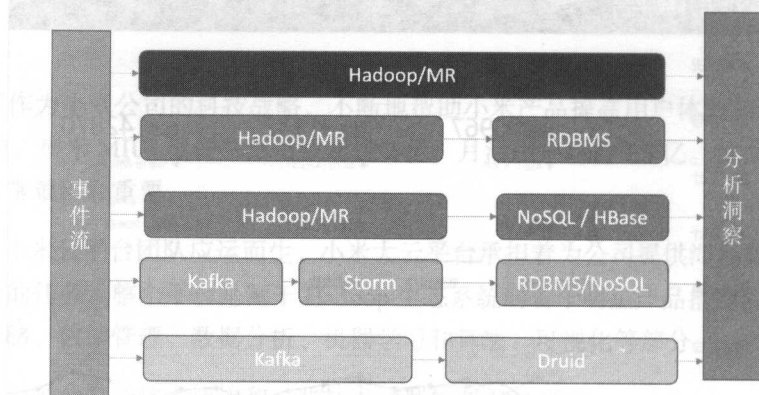


图 10-3 小米统计数据流

Druid 作为一种实时分析数据库，提升了小米大数据平台和商业产品部门的实时数据分析能力。

10.1.2 场景二：广告平台实时数据分析

Druid 来源于广告业务，小米广告平台也利用 Druid 进行实时的数据分析，帮助实时分析线上的各种维度的变化，包括上线部署的实时监控分析、A/B 测试的效果查询、一些细粒度的数据分析。

对广告数据有两条路径进行处理：一条是实时的数据流，通过 Druid 处理，主要是针对内部的实时数据分析需求；另一条是通过 Mini-batch 方式。

数据的 DataSource（数据源）包括：

- 小米广告交易平台（Xiaomi Ad Exchange, MAX）：广告流量的调度管理平台。
- 广告平台的计费分析模块：广告主的计费、各种维度数据。
- 广告媒体分析数据：各个广告媒体的请求、展现等数据。

比如对于广告计费分析模块，Druid 会包括实时的广告主计费信息，这些数据用于内部的数据分析，不用于广告主投放平台。广告主投放平台使用 Mini-batch 方式，通过可重放的方式更新和聚合数据结果。

在使用 Druid 的过程中也会碰到一些问题。

1. 关于查询界面

Druid 的查询语言还不是特别友好，在第一阶段部署 Druid 后，我们开发了一套 Druid 查询接口，主要是满足业务的需求，初期效果还好，但是随着数据源的增加，每次增加数据源都需要额外开发一些界面，增加维度，也需要修改前端工程，因此效率也不高。在后期的工作中，尝试了Pivot 工具，功能使用方便，渐渐代替了自定义的查询界面。

2. 关于查询效率

Druid 的大部分时间性能表现都很好，但是如果进行长时间范围的查询，系统会变得非常慢。为了解决这个问题，对于频繁查询的数据源，可以分为两个部分：一部分是按照分钟级别聚合的数据源，数据保持 10 天；另一部分是按照小时级别聚合的数据源，数据保持 2 年。每天晚上的时候，聚合小时级别的数据，这样可以避开高负载的集群时间。聚合粒度与查询效率的关系如图 10-4 所示。

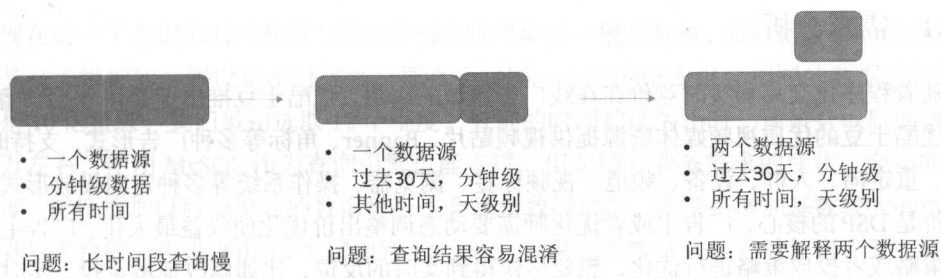


图 10-4 聚合粒度与查询效率的关系

3. 部署情况

Druid 集群每天处理近百亿的事件请求，集群规模为近 10 台机器，索引服务和历史节点数量相当，机器的数量随着事件数的增长而增加。当数据源在某时间数据急剧增加时，系统索引文件所占用的 CPU 会很高，有时候影响正常的查询性能。

第一阶段，我们尝试在服务层使用流量控制，但是后来放弃了。原因是，数据在 1 小时后有过期机制，因此如果有数据无法进入系统，那么这些数据可能丢失。因此，我们还是尽量让数据进入 Druid 系统，虽然偶尔会带来系统的峰值压力。

基于 Druid 的架构和数据流如图 10-5 所示。

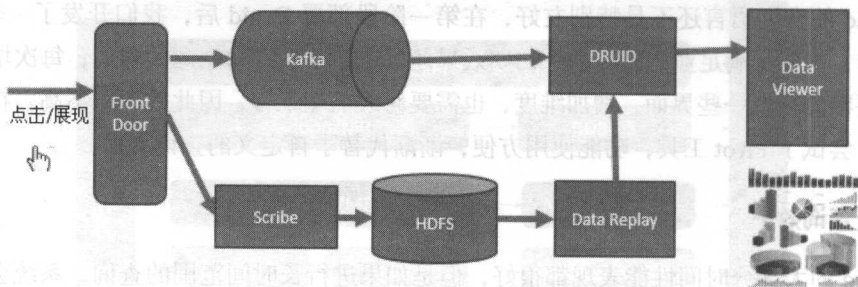


图 10-5 基于 Druid 的架构和数据流

10.2 优酷土豆

10.2.1 需求分析

随着程序化交易和实时竞价在在线广告领域的应用，优酷土豆推出了 DSP 平台“睿视”，依托优酷土豆的优质视频媒体资源提供视频贴片、Banner、角标等多种广告形式，支持时间、地域、重定向、人群、设备、频道、视频标签、浏览器、操作系统等多种定向投放形式。实时竞价是 DSP 的核心，广告主或者优化师需要动态调整出价优化使收益最大化。广告主调整出价策略或者投放策略进行优化，想要尽快得到实时的反馈，比如修改地域定投，实时分地域查看竞得率和转化率，甚至是分钟粒度的。睿视平台提供了广告主、投放、素材、广告位、地域等 17 个维度，出价数、曝光数、独立访客数、点击数、落地页的相关转化指标以及扣费情况等 22 个统计项。不同数据量级的技术架构不同，数据增大一个量级就会给技术架构带来很大的挑战。简单一点的，数据日增量在百万量级或千万量级，使用 Kafka、Storm 进行实时计算，MySQL 做 OLAP 引擎就可以支撑。我们来看一下睿视平台的需求以及技术挑战。

- 数据日增量为 10 亿量级。
- 任意多维钻取分析，17 个维度、22 个统计项。
- 高并发下的交互式查询，给广告主提供在线查询服务。
- 数据延迟在分钟级别，导入即可被查询。
- 多维度下的独立访客统计，独立访客数在千万量级。

10.2.2 技术选型及工程实践

从需求背景来看，我们面临的技术挑战是海量数据、交互式查询以及实时性。影响技术选型的首要因素是海量数据，需求中提到的 10 亿量级是最细粒度的原始数据，在 OLAP 场景中一般会进行预聚合来减少数据量。影响预聚合以后的数据量的因素如下。

- 维度的个数
- 维度的基数
- 维度组合的稀疏程度
- 查询的时间粒度

这里提到了基数的概念，基数是维度中不重复值的数量。用户维度是一个高基数的维度，如果在数据模型中添加用户维度，那么预聚合的量级至少是千万级，维度组合越稀疏，预聚合的效果就越好。

现在讲一下查询的时间粒度。原始数据的时间单位一般是毫秒，而数据分析一般是分钟、小时甚至天级别的。假设是小时级别，那么会对同一小时内维度组合相同的数据进行聚合，同样是 10 亿量级，我们最初选取 12 个维度，查询的时间粒度为小时，预聚合以后的数据只有百万左右，使用 MySQL 作为查询引擎也能支撑。但是随着业务需求的变化，会不断地增加维度，维度增加以后预聚合的数据量级也会呈指数级增加，特别是增加基数较大的维度以后。睿视平台的分析维度由 12 个增加到 17 个时，预聚合的数量从百万增长到千万，MySQL 的扩展性不足以支持指数级增长。

很多 SQL On Hadoop 支持海量数据，能做到交互式查询，但却不支持实时性。后来我们调研了 Druid，Druid 本身就是为统计而生的，它诞生的背景也是为互联网广告公司提供实时多维分析。睿视平台使用 Druid 以后的整体架构示意图如图 10-6 所示。

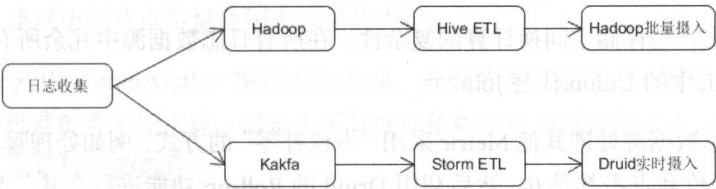


图 10-6 睿视平台使用 Druid 以后的整体架构示意图

这是典型的 Lamdba 架构，分为实时和离线两部分。数据源是日志收集系统收到的日志文件采用 tsv 格式存储，实时部分采用 Kafka 和 Storm 组合计算增量数据，离线部分采用 Hadoop Indexer 批量生成 Segment。Druid 的架构组合方式非常灵活，可以采用以下方式。

- 只使用实时方式，使用索引服务或者实时节点实时摄入数据。
- 只使用批量方式，使用 Hadoop Indexer 批量摄入数据。
- 使用批量和实时相结合的方式。

由于日志收集以及实时计算过程会出现数据丢失情况，所以我们采用批量和实时相结合的方式。在第二天凌晨时启动 Hadoop Indexer 批量生成 Segment 去覆盖当天实时生成的 Segment，Druid 的 Segment 是有版本的，Coordinator 一旦发现有高版本的 Segment，就会通知历史节点去加载高版本，同时删除低版本的 Segment，默认采用 Segment 的生成时间作为版本。

再看具体业务。睿视系统的竞价、曝光、点击等效果数据是分日志文件存储的，而 Druid 只支持单表，那么我们就需要预先将竞价、曝光、点击这三种数据源连接到一张表中。离线部分连接操作还好处理，可以通过 Hive 的连接功能将三种效果数据合并到一起，但实时数据要实现两个数据流之间的连接操作就较为复杂，一般实现有两种方案。

- 采用同样的唯一 id 将不同的日志关联起来。
- 在不同的日志中冗余所有的维度。

第一种方案，例如竞价事件是所有事件产生的源头，在竞价日志中记录唯一 id（叫作 RequestId）和所有的维度，当该竞价事件产生了曝光和点击时，则只需要在曝光和点击中记录 RequestId，而不需要记录相关的维度，然后使用 RequestId 关联起来。这种方式的优点是不冗余数据，减少了数据的存储。分布式日志收集的特点是无序，虽然竞价事件一定是最先发生的，但无法保障总是竞价事件先收集到，假设曝光事件先到，那么此时就需要在实时流计算过程中缓存曝光事件，直到具有相同 RequestId 的竞价事件到来。鉴于此，需要能够处理状态的实时流计算框架来实现数据流之间的 Join，Apache 开源的 Samza 和 Flink 都能解决上述问题。

第二种方案，用存储空间换计算的复杂性。在所有日志数据源中冗余所有的维度，可以采用类似于 SQL 中的 Union 代替 Join。

不同的日志数据源处理其他 Metric 采用“占位补零”的方式，例如处理曝光日志，Metric 记作 0, 1, 0。竞价和点击都是 0，然后利用 Druid 的 Roll-up 功能进行合并。为了减轻 Druid 的数据摄入压力，Storm 在一定时间内进行预先合并。同样采用占位补零的方式，使用维度组合作为 key，根据 key 对维度进行分组，使相同维度组合的事件落在同一 Join Bolt 中，然后在 Join Bolt 中按照 Key 合并不同数据流的事件，定时发送到 Druid，如图 10-7 所示。

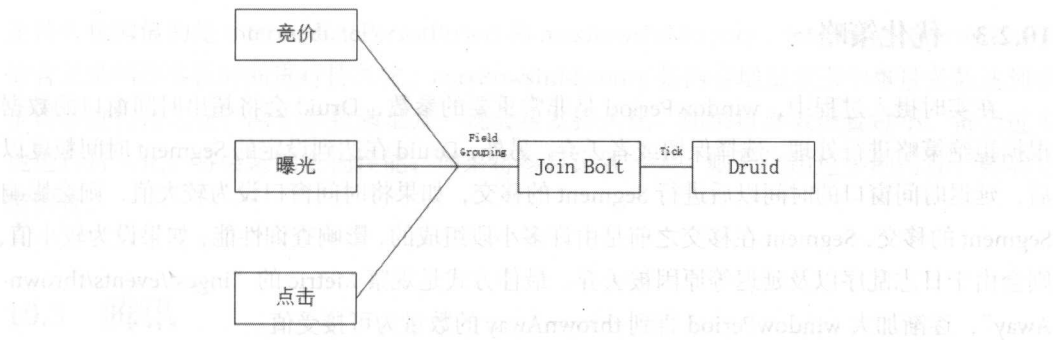


图 10-7 Storm 与 Druid 集成示意图

节点类型	数量	配置
实时节点	4 台	24 核，64GB 内存
历史节点	8 台	24 核，64GB 内存
协调节点	2 台	12 核，32GB 内存
查询节点	2 台	12 核，32GB 内存

实时部分我们仍然采用 Druid 的早期架构实时节点，其优点是足够简单，仅依赖于 Kafka 等消息队列，当内存增量索引持久化以后再提交 Offset 保障数据不会丢失；缺点是没有副本，以及修改 Segment 以后需要重启。其最大的问题是没有副本不能保障高可用性，所以作者建议使用索引服务（Indexing Service）。

历史节点并没有使用 SSD，是因为目前睿视平台的索引还比较小，几乎在内存中都能覆盖。历史节点可以调整 maxCacheSize 参数来决定 Segment 在内存和磁盘中的覆盖比例，这也是历史节点的容量最大值，如调整成和内存值一样能避免磁盘 IO，但同时也意味着需要更多的历史节点，建议历史节点尽量使用大容量的内存。

由于 Broker 采用 Scatter/Gather 模式进行查询，查询节点只是汇聚查询结果，所以其负载相对低一点，机器配置不用太好，但是由于其性能依赖于可用内存的大小，因此官方推荐尽量为查询节点配置较大的内存。

使用 Druid 以后的架构，其性能表现良好，通过将 Druid 的 Metric 发送到 OpenFalcon（开源监控系统）中监控统计到每分钟内最大的响应时间大部分在 1 秒以内，平均响应时间在 300 毫秒左右，数据从发生到导入 Druid 的延迟时间在 2 分钟以内。

10.2.3 优化策略

在实时摄入过程中, `windowPeriod` 是非常重要的参数, Druid 会将超出时间窗口的数据根据拒绝策略进行处理, 选择保留或者丢弃。另外, Druid 在达到设定的 `Segment` 时间粒度以后, 延迟时间窗口的时间以后进行 `Segment` 的移交, 如果将时间窗口设为较大值, 则会影响 `Segment` 的移交, `Segment` 在移交之前是由许多小段组成的, 影响查询性能; 如果设为较小值, 则会由于日志乱序以及延迟等原因被丢弃。最佳方式是观察 `Metric` 的 “`ingest/events/thrown-Away`”, 逐渐加大 `windowPeriod` 直到 `thrownAway` 的数量为可接受值。

拒绝策略的可选项有 `messageTime`、`serverTime` 和 `none` 三种。`messageTime` 和 `serverTime` 两种策略在超出时间窗口以后都会被丢弃, 不同的是时间窗口的选择, `messageTime` 是当前事件的时间戳和已处理事件中最大的时间戳进行比较, `serverTime` 是当前事件的时间戳和当前系统时间戳进行比较, `messageTime` 仅会因为事件乱序导致被丢弃, 而不会因为延迟被丢弃; 而 `serverTime` 是两者都有可能。由此看来, `messageTime` 更适合一些, 但是因为 `messageTime` 会导致 `Segment` 的移交延迟, 如日志收集存在延迟, 则会出现只能移交 T-2 的 `Segment`, 所以在实时节点生产环境中应尽量使用 `serverTime`。`none` 表示不丢弃, 在实时节点模式下会导致 `Segment` 不能移交, 所以不适合设置成 `none`。但如果采用索引服务的话, 它却是最佳选择, 在索引服务中会创建任务读取设定时间段的数据, 不在设定时间段内的数据依然会被丢弃, 但乱序的数据则不会丢弃。当达到结束的时间点以后, 结束读取并且移交, 而不像实时节点那样, `Segment` 的移交受拒绝策略影响。

`SegmentGranularity` 和 `QueryGranularity` 如何配置是初学者比较迷茫的问题之一。首先来明确这两个参数的含义, `QueryGranularity` 是查询的最小时间粒度, 它主要影响数据摄入过程和查询过程中的 Roll-up, `QueryGranularity` 越大, Roll-up 以后的数据量越小, 查询越快, 但是同时也意味着不能查询更细时间粒度的数据。`SegmentGranularity` 是 `Segment` 分段的时间粒度, 比如 `SegmentGranularity` 为 `hour`, 那么说明 `Segment` 是按照小时分段的, 每个 `Segment` 存储 1 小时的数据。通常, 作者建议 `SegmentGranularity` \geq `QueryGranularity`, 但这并不是必须遵守的。例如有一个 `DataSource` 的 `QueryGranularity` 为 `day`, 采用实时摄入的方式, 如果遵守上述原则, `SegmentGranularity` 至少是 `day`, 那么一天之内 `Segment` 都不能移交给历史节点, 导致查询性能下降。

经过前面章节的学习, 我们知道 Druid 的实时部分采用类似于 LSM-tree 的架构。事件数据首先在内存增量索引中累积, 此过程中会进行 Roll-up。当达到一定阈值以后, 采用异步线程将内存增量索引持久化到磁盘中, 持久化后的索引采用倒排和 Bitmap 索引, 适合查询, 当达到 `SegmentGranularity` 设定的条件时, 将所有的持久化索引合并成一个 `Segment` 并移交。决

定持久化阈值的是 `intermediatePersistPeriod` 和 `maxRowsInMemory`。`intermediatePersistPeriod` 的含义是间隔多长时间进行持久化，`maxRowsInMemory` 是内存增量索引中事件条数达到多少以后进行持久化，两个参数满足其一就会触发持久化。如果将参数设置过小，将会过多地进行持久化，将会影响查询性能；如果将参数设置过大，则会使用过多的内存，影响垃圾回收。

10.3 腾讯

数据分析是企业级 SaaS 服务中重要的一块，作为业界领先的 SCRM 产品，腾讯企点通过数据分析让企业更加了解和认识他们的客户并从中分析出客户转化、留存等一些关键指标，进一步驱动产品迭代和精细化运营。随着互联网中各类海量的交互、访问数据，数据分析面临着新的挑战，其特点如下。

- 数据的快速收集与统计处理：一方面，需要收集和处理的的数据量大（处理用户访问的全量行为数据，每日上亿规模的数据量）；另一方面，需要计算分析出统计意义上的数值结果。
- 时效性与灵活性：需要及时获得分析结果，并能灵活地在多个维度上进行组合选取对比。

10.3.1 工程实践

1. MySQL/PostgreSQL

方案：离线定期做数据聚合统计处理。将多个分析维度做排列组合，聚合统计结果并按处理的周期进行划分存储（如划分为天表、小时表）。

- 优点：服务稳定且使用简单（SQL 标准化使用），由于存储的是面向业务统计的结果集，因此数据量大大减少。基于传统 OLAP 数据库可直接面向业务层查询使用。
- 缺点：业务定制化导致损失了灵活性，且统计区段固定导致无法解决跨分区统计问题（比如对两天内的访问用户做去重统计），其只能解决统计指标进行并集操作的统计需求。因为数据依赖离线任务，故在时效性上有延迟。

2. Elasticsearch (ES)

方案：访问数据进行 ETL 后入 ES，写支持实时和离线两种方式入库，查询时根据不同的维度进行聚合统计查询。

- 优点：ES 在数据查询上更灵活且集群部署简单（外部依赖少），由于 ES 存储了入库时的原始数据，故在业务上可以支持更细粒度的查询，比如查询明细流水数据。
- 缺点：资源更多，需要更多的机器。在数据量大的聚合统计方面查询延迟与并发表现不如 Druid。

3. Druid

方案：根据事先确定好的分析维度和统计指标，利用 Spark 任务清洗出干净的数据集写入 Druid。由于查询的粒度最小为小时，故可以按照小时级别的粒度在入库时进行 Roll-up 处理。聚合统计数据将大大减少 Druid 中存储的数据量。以统计去重后的客户数指标为例，由于 Druid 的聚合统计 HyperUnique 采用了 HyperLogLog 算法，而 HyperLogLog 算法是支持并集操作的，因此即使是计算跨小时的时间范围统计也能很好地得到支持。

- 优点：通过入库时的预聚合，待统计分析的数据量大大减少，降低了存储成本，提高了查询效率。Druid 支持多种聚合统计估算算法，在应对大数据量分析场景中有着上佳的表现。同时，由于 Druid 在设计上采用 shared-nothing 架构，故其在并发查询上也有着优异的表现。
- 缺点：外部依赖较多且运维部署麻烦。由于在入库时采取预聚合计算的策略，故原始信息的缺失导致无法支持明细数据查询。

结合业务在技术方案的选取上有如下考虑。

- 访客上报的数据量大，每日上报记录数至少都是上亿规模。
- 需要支持任意时间范围的统计，以及灵活地对比不同的维度数据。
- 需要支持更大的并发查询（企业级应用一般需支持上百 QPS 的服务要求）。

首先，由于需要在任意时间范围做统计分析，故 MySQL/PostgreSQL 存储结果集的方案不能满足需求，且随着数据量的日益增长，传统 OLAP 存储系统在扩展性方面也面临挑战。其次，从统计分析功能上看，ES 和 Druid 都能满足需求，但这里需要考虑使用上的权衡。由于可以事先确定待分析的维度和统计指标，相比数据入库至 ES 时存储原始数据，数据入库至 Druid 时可以进行 Roll-up 聚合处理，使得其有更低的存储成本和更高的查询效率。因此，针对统计分析数据量大（上亿规模）、维度多（待分析维度确定）和分析结果集小且为统计数值的需求采用 Druid 方案更为合适。Druid 分别通过入库 Roll-up（减少数据量）、列式存储（支持灵活的维度）、统计指标聚合计算等来实现在统计分析方面的优化。Druid 支持如下近似算法，满足了大数据量、高基数等方面的分析需求。

- 近似直方图和分位数。
- HyperUnique, 利用 HyperLogLog 算法求基数, 比如计算访客的 UV 数。
- DataSketch, 相比 HyperLogLog 算法 (只支持并集计算), 其支持集合运算如交集、并集、差集, 比如计算同时访问过网站 A 和网站 B 的访客 UV 数。

基于 Druid 的统计分析系统架构图如图 10-8 所示。

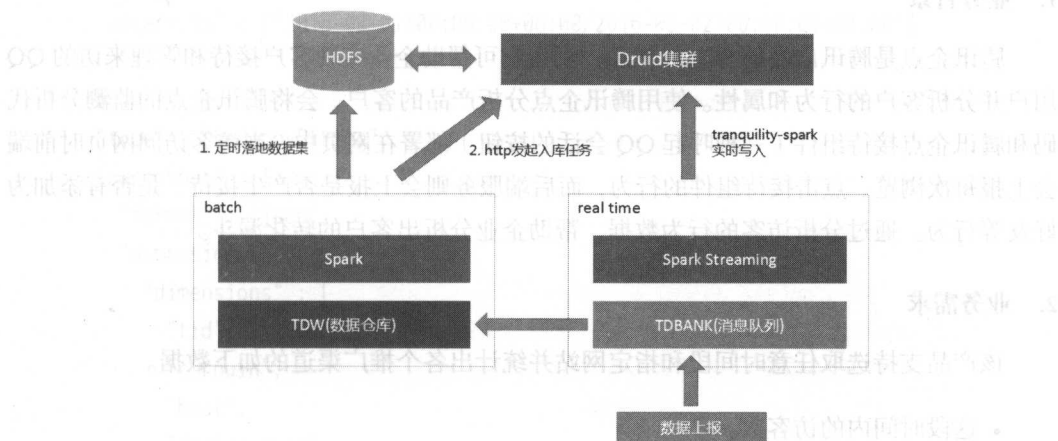


图 10-8 基于 Druid 的统计分析系统架构图

从架构图中可以看出, 在工程实践中使用 Druid 包含实时和离线两部分。

(1) 实时部分

利用 Spark Streaming 消费消息队列中上报的数据并进行业务层面的数据处理后入库至 Druid。实时部分可满足对数据时效的需求, 通常 Segment 数据文件根据业务情况划分为分钟至小时粒度。但由于实时部分只能进行数据追加写操作, 且数据上报延迟等原因, 因此在实时部分的完整性上有一定的缺失, 需要离线部分的处理来完善数据。

(2) 离线部分

通常以天粒度的批量任务方式对数据进行修正和补全。每日数据上报落地至数据仓库后, 启动 Spark 离线任务批量处理数据并写入 Druid 依赖的 HDFS 路径, 完成后向 Druid 集群发起离线入库任务 (Druid 根据任务 Schema 启动相应的 MapReduce 任务对数据做入库处理)。由于 Druid 存储数据时以 Segment 为粒度, 因此离线入库可以根据时间范围对实时部分写入的 Segment 进行替换, 完成对实时部分数据的修正和补全 (这里有个优化处理, 在业务层面对历史数据的分析粒度只需要到天级别, 而实时部分则是按小时划分的, 那么离线任

务按天级别对前一天的数据做天粒度的 Roll-up 处理可进一步压缩存储数据)。若增加了数据源维度,或者相关字段的计算方式发生了变化,那么也可以利用离线方式对历史数据进行修复,即离线部分保障了 Druid 中的数据总是可以被重新计算和恢复。

10.3.2 业务实践

1. 业务背景

腾讯企点是腾讯出品的 SCRM 产品,利用它可帮助企业实现客户接待和管理来访的 QQ 用户并分析客户的行为和属性。使用腾讯企点分析产品的客户,会将腾讯企点的监测分析代码和腾讯企点接待组件(一种呼起 QQ 会话的按钮)部署在网页中。当访客访问网页时前端会上报每次浏览、点击接待组件的行为,而后端服务则会上报是否产生接待、是否有添加为好友等行为。通过分析访客的行为数据,帮助企业分析出客户的转化漏斗。

2. 业务需求

该产品支持选取任意时间段和指定网站并统计出各个推广渠道的如下数据。

- 这段时间内的访客数。
- 在访问的客户中点击接待组件的访客数。
- 有多少访客点击接待组件时不在好友表中。
- 有多少访客被转化为好友,并且每一步都需要支持统计新老访客数。

上述统计数据从大到小、从粗到细形成漏斗状,因此被称为客户分析漏斗。

3. 实现方式

由于每日上报的数据来源较多,且数据之间在实时性匹配上也存在着困难,故我们采用离线方式,即每日离线清洗处理数据后批量写入 Druid。离线任务 Schema 定义如下:

```
{  
  "type" : "index_hadoop",  
  "spec" : {  
    "ioConfig" : {  
      "type" : "hadoop",  
      "inputSpec" : {  
        "type" : "static",  
        "paths" : "hdfs://${集群地址}/olap/visitor_stat/"  
      }  
    }  
  }  
}
```

```

},
"dataSchema" : {
  "dataSource" : "visitor_statistics",
  "granularitySpec" : {
    "type" : "uniform",
    "segmentGranularity" : "day",
    "queryGranularity" : "day",
    "intervals" : ["2016-08-01T00:00:00+08:00/2016-08-02T00:00:00+08:00"]
  },
  "parser" : {
    "type" : "hadoopString",
    "parseSpec" : {
      "format" : "json",
      "dimensionsSpec" : {
        "dimensions" : [
          "tid",
          "corpuin",
          "host",
          "device_type",
          "is_new",
          "is_ad",
          "is_sem",
          "ad_source",
          "ad_media",
          "ad_campaign",
          "ad_term",
          "ad_content",
          "is_click",
          "in_customerdb_before_click",
          "in_customerdb_after_talk"
        ],
        "dimensionExclusions" : []
      }
    },
    "timestampSpec" : {
      "format" : "auto",
      "column" : "timestamp"
    }
  }
}

```

```

    },
    "metricsSpec" : [
      {
        "name" : "count",
        "type" : "count"
      },
      {
        "name" : "uv",
        "type" : "hyperUnique",
        "fieldName" : "qidianid"
      }
    ]
  },
  "tuningConfig" : {
    "type" : "hadoop",
    "partitionsSpec" : {
      "type" : "hashed",
      "targetPartitionSize" : 5000000
    },
    "maxRowsInMemory" : 100000,
    "cleanupOnFailure" : false,
    "jobProperties" : { // MapReduce任务优化参数
      "mapreduce.map.memory.mb" : 2048,
      "mapreduce.map.java.opts" : "-server -Xmx1536m -Duser.timezone=UTC+0800 -Dfile.encoding=UTF-8",
      "mapreduce.reduce.memory.mb" : 6144,
      "mapreduce.reduce.java.opts" : "-server -Xmx2560m -Duser.timezone=UTC+0800 -Dfile.encoding=UTF-8",
      "mapreduce.job.reduces" : 21,
      "mapreduce.job.jvm.numtasks" : 20,
      "mapreduce.reduce.shuffle.parallelcopies" : 50,
      "mapreduce.reduce.shuffle.input.buffer.percent" : 0.5,
      "mapreduce.task.io.sort.mb" : 250,
      "mapreduce.task.io.sort.factor" : 100,
      "mapreduce.jobtracker.handler.count" : 64,
      "mapreduce.tasktracker.http.threads" : 20,
      "mapreduce.output.fileoutputformat.compress" : false,
      "mapreduce.output.fileoutputformat.compress.type" : "BLOCK",

```

```

"mapreduce.output.fileoutputformat.compress.codec": "org.apache.hadoop.io.
  compress.DefaultCodec",
"mapreduce.map.output.compress": true,
"mapreduce.map.output.compress.codec": "org.apache.hadoop.io.compress.
  DefaultCodec",
"mapreduce.map.speculative": false,
"mapreduce.reduce.speculative": false,
"mapreduce.task.timeout": 1800000
}
}
}
}

```

从上述 Schema 可以看出,我们将离线清洗后的数据存入 HDFS,再将入库任务导入 Druid,每一条入库的记录表示一次访问。其中, `is_new` 表示是否为新访客, `is_click` 表示是否点击了接待组件, `in_customerdb_before_click` 表示点击时是否是好友, `in_customerdb_after_talk` 表示该次访问在不是好友的情况下是否被转化为好友。这几个字段的取值为 0: 否, 1: 是。 `ad_source`、`ad_media`、`ad_campaign` 等表示各个广告推广渠道。

虽然经过了数据清洗,但是由于上报数据时每一条访问记录都要存储,且使用腾讯企点分析服务的客户众多,每个客户的访客和点击数据都入库将导致原始数据集十分庞大。但是由于使用了 Druid 来统计各个阶段的访客数,在每日数据入库时做了 `roll-up` 处理后数据量大大减少。灰度阶段每天约 1.2GB 的原始数据经过聚合处理后只有 200KB 左右,同时相应的统计指标因为入库时做了预处理,故查询性能也十分优异。

通过 `groupBy` 方式进行统计查询,查询语句如下:

```

{
  "aggregations": [
    {
      "fieldName": "uv",
      "name": "uv",
      "type": "hyperUnique"
    }
  ],
  "dataSource": "visitor_statistics",
  "dimensions": [
    "ad_source",
    "ad_campaign",

```



```

    "ad_media",
    "is_new",
    "is_click",
    "in_customerdb_before_click",
    "in_customerdb_after_talk"
  ],
  "filter": {
    "fields": [
      {
        "dimension": "host",
        "type": "selector",
        "value": "xxx.xxx.com"
      },
      {
        "dimension": "corpuin",
        "type": "selector",
        "value": "xxx"
      }
    ],
    "type": "and"
  },
  "granularity": "all",
  "intervals": [
    "2016-08-20T00:00:00.000Z/2016-08-25T00:00:00.000Z"
  ],
  "queryType": "groupBy"
}

```

查询结果如下：

```

[[
  {
    "version": "v1",
    "timestamp": "2016-08-20T00:00:00.000Z",
    "event": {
      "is_new": "0",
      "ad_media": null,
      "uv": 18351.85859471761,
      "in_customerdb_after_talk": "0",
      "in_customerdb_before_click": "0",

```

```

        "is_click" : "0",
        "ad_source" : null,
        "ad_campaign" : null
    }
}
]]

```

用于 groupBy 的字段虽然有 7 个，但是由于每个字段的值域（基数）很小，因此最后查出来的结果集并不大。最终业务侧对查询结果集进行适当处理，即可得出漏斗。除了各个推广渠道的漏斗，还需要统计汇总信息，那么只需要将 groupBy 中的 ad_source、ad_campaign、ad_media 去掉即可，在此不再复述。

4. 优化方案

该业务涉及多层漏斗的计算，虽然应用层可以利用 Group By 操作筛选出所有组合后再做处理，但是在查询效率和使用上仍然有改进的空间。即：离线清洗每条记录时，可根据标识字段分别洗出多份冗余的 ID 字段（is_click -> click_wpa_qidianid, in_customerdb_before_click -> before_click_indb_qidianid, in_customerdb_after_talk -> after_talk_indb_qidianid），然后将这些 ID 字段在 Druid 入库时分别设置对应到不同的 Metric 上，这样一次查询即可直接统计分析漏斗各层对应的数值。

优化后的入库 Schema 定义如下：

```

{
    "type" : "index_hadoop",
    "spec" : {
        "ioConfig" : {
            "type" : "hadoop",
            "inputSpec" : {
                "type" : "static",
                "paths" : "hdfs://${集群地址}/olap/visitor_stat/"
            }
        }
    },
    "dataSchema" : {
        "dataSource" : "visitor_statistics",
        "granularitySpec" : {
            "type" : "uniform",
            "segmentGranularity" : "day",
            "queryGranularity" : "day",
            "intervals" : ["2016-08-28T00:00:00+08:00/2016-08-29T00:00:00+08:00"]
        }
    }
}

```

```

},
"parser" : {
    "type" : "hadoopyString",
    "parseSpec" : {
        "format" : "json",
        "dimensionsSpec" : {
            "dimensions" : [
                "tid",
                "corpuin",
                "host",
                "device_type",
                "is_new",
                "is_ad",
                "is_sem",
                "ad_source",
                "ad_media",
                "ad_campaign",
                "ad_term",
                "ad_content"
            ],
            "dimensionExclusions" : []
        },
        "timestampSpec" : {
            "format" : "auto",
            "column" : "timestamp"
        }
    }
},
"metricsSpec" : [
    {
        "name" : "count",
        "type" : "count"
    },
    {
        "name" : "visit_count",
        "type" : "hyperUnique",
        "fieldName" : "visit_qidianid"
    }
],

```

```

{
  "name" : "click_wpa_count",
  "type" : "hyperUnique",
  "fieldName" : "click_wpa_qidianid"
},
{
  "name" : "before_click_indb_count",
  "type" : "hyperUnique",
  "fieldName" : "before_click_indb_qidianid"
},
{
  "name" : "after_talk_indb_count",
  "type" : "hyperUnique",
  "fieldName" : "after_talk_indb_qidianid"
}
]
},
"tuningConfig" : {
  "type" : "hadoop",
  "partitionsSpec" : {
    "type" : "hashed",
    "targetPartitionSize" : 5000000
  },
  "maxRowsInMemory" : 100000,
  "cleanupOnFailure" : false,
  "jobProperties" : {...}
}
}
}

```

查询语句如下：

```

{
  "queryType": "groupBy",
  "dataSource": "visitor_stat_funnel",
  "granularity": "all",
  "dimensions": ["ad_source", "ad_media", "ad_campaign", "ad_term"],
  "filter": {
    "type": "and",

```

```

    "fields": [
      { "type": "selector", "dimension": "host", "value": "wap.qidian.qq.com" }
    ],
    "aggregations": [
      { "type": "longSum", "name": "count", "fieldName": "count" },
      { "type": "hyperUnique", "name": "visit_count", "fieldName": "visit_count" },
      { "type": "hyperUnique", "name": "click_wpa_count", "fieldName": "click_wpa_count" },
      { "type": "hyperUnique", "name": "before_click_indb_count", "fieldName": "before_click_indb_count" },
      { "type": "hyperUnique", "name": "after_talk_indb_count", "fieldName": "after_talk_indb_count" }
    ],
    "intervals": ["2016-08-28T00:00:00+08:00/2016-08-29T00:00:00+08:00"]
  }
}

```

查询结果如下：

```

[ {
  "version" : "v1",
  "timestamp" : "2016-08-27T16:00:00.000Z",
  "event" : {
    "visit_count" : 3833.9509193395515,
    "count" : 6996,
    "click_wpa_count" : 77.44604364148258,
    "before_click_indb_count" : 75.37001710790979,
    "after_talk_indb_count" : 66.05385244590596
  }
} ]

```

聚合器对比：在上例的优化过程中，由于涉及基数统计，故可以考虑使用 HyperUnique 和 ThetaSketch 两种聚合器。在去重统计基数方面，ThetaSketch 相比 HyperUnique 可以支持多个筛选条件的交集做去重统计，但是 ThetaSketch 需要耗费更多的存储空间。在具有相同原始数据集的情况下，入库一天的数据存储占用情况为：原始数据集占用 1.2GB；ThetaSketch 占用 3.83MB；HyperUnique 占用 500KB。因此，需要结合业务的使用情况和存储空间来选择聚合器。

10.4 蓝海讯通

创立于2008年的蓝海讯通科技股份有限公司（OneAPM）是一家科技型的创业公司，专注于企业应用性能管理（Application Performance Management, APM）与企业级信息系统运维管理（Information Technology Operation Management, ITOM）领域，目前在国内处于行业领先地位。从技术上讲，OneAPM的产品主要是通过字节码工具（ByteCode Instrument, BCI）或数据包捕获等技术，获取软件与系统在运行时产生的各种性能数据或指标，从而对应用程序的性能、可用性及质量进行分析，最终实现对IT基础架构的性能进行管理。不难看出，OneAPM技术解决方案的部分数据（主要是指标数据）来源及其处理要求具有如下几个主要特征。

- 数据主要由机器或软件系统产生，且频率较高，是典型的大数据。
- 数据有很明显的时序特征。
- 数据处理要求能及时、准确完成，否则难以对其所监控的系统进行有效管理。

鉴于上述数据特点，OneAPM的数据解决方案中包含了适合其应用场景的Druid项目，并且在其核心的指标数据（Metric Data）处理模块中利用Druid成功地完成了对海量时序数据的实时处理。目前，OneAPM主要有三个产品线在同时使用Druid。

- Application Insight（Ai）：一个贯穿应用系统全生命周期的真实用户体验管理和应用性能管理平台级解决方案，主要应用在服务器端应用性能监控与管理上。
- Browser Insight（Bi）：基于真实用户的Web前端性能监控平台，主要用于统计分析网站流量，定位网站性能瓶颈，并且支持浏览器、微信、App浏览HTML和HTML5页面。
- Mobile Insight（Mi）：一个真实可见的移动应用性能监控系统，能够为用户提供移动应用性能测试。

目前，上述三个产品使用Druid的大致架构体系相同：通过自己产品的探针来实时采取监控对象的指标数据，然后通过网络将包含指标数据的数据包发送到后端集群进行处理。后端集群通过Nginx来接收这些指标数据，然后通过多个数据收集器（Data Collector）对指标数据的数据包进行解压缩，接下来立刻将解压缩后的数据存放到作为中转缓存的Kafka集群。紧接着，多个对数据进行逻辑处理的数据消费者（Data Consumer）会根据既定逻辑拉取并处理数据，在完成数据处理后会立刻将数据打回到Kafka集群中的另外一个供Druid消费的Topic，然后Druid集群就会通过其Kafka Firehose来拉取并消费数据，最终将指标数据都聚合存放到Druid集群中以供后端分析应用查询使用。

虽然这三个产品使用 Druid 的大致架构体系相同，但是它们各自的 Druid 集群却采取了不同的数据消费方式。

- Druid 实时节点：Ai, Bi

- Druid 索引服务：Mi

除了使用不同的数据消费方式外，每个产品对 Druid 的各个使用细节也不尽相同，比如 DataSource Schema 的定义与使用 Query 的种类等。由于篇幅有限，这里就不再详细描述所有的 Druid 使用细节了，仅仅描述其中一个重要的使用特点——金字塔式的 DataSource 存储方案（本文中均简称为“金字塔方案”）。

在 OneAPM 公司基于 Druid 的解决方案中，主要是 Ai 这个产品使用到了金字塔方案：它用三个不同的 Druid DataSource 来消费同一个 Kafka Topic，并且这三个 Druid DataSource 的 Schema 也基本相同，仅仅是数据的聚合粒度不同——通过 queryGranularity 进行设置，其聚合粒度分别是 1 分钟、10 分钟和 1 小时。为了方便介绍，我们为这三个 DataSource 分别取名为：

- druid_metric，聚合粒度是 1 分钟。

- druid_metric_10minute，聚合粒度是 10 分钟。

- druid_metric_1hour，聚合粒度是 1 小时。

有读者看到这里估计会感到奇怪：为什么要对同一个 Kafka Topic 创建三个独立但 Schema 却大致相同仅聚合粒度不同的 Data Source？其实大家的疑惑是有道理的，毕竟如果仅仅是为了满足不同聚合粒度的查询（比如上述的 1 分钟、10 分钟和 1 小时等），其实完全可以在 Druid 集群中只创建一个能满足最小查询粒度的 DataSource（如 1 分钟的 DataSource）即可，因为 Druid 支持在 DataSource 上执行大于或等于其查询粒度即 queryGranularity 的聚合查询，但不支持执行小于其查询粒度的聚合查询——也就是说，Druid 在聚合消费数据并存入集群后数据的最小精度便固定了，而低于该精度的细节也都已经丢失了。这是 Druid 的一个基本且重要的特点。下面举个简单的例子，假如我们想执行聚合粒度为 10 分钟的查询，如：

```
{
  "queryType": "groupBy",
  "dataSource": "{DATASOURCE_NAME}",
  "granularity": {
    "type": "duration",
    "duration": 60000,
    "origin": "2015-11-09T02:00:00"
```

```
},
"dimensions": [
  "applicationId",
  "metricId"
],
"aggregations": [
  {
    "type": "longSum",
    "name": "num1",
    "fieldName": "num1"
  }
],
"filter": {
  "type": "and",
  "fields": [
    {
      "type": "selector",
      "dimension": "applicationId",
      "value": "1007323"
    },
    {
      "type": "selector",
      "dimension": "metricId",
      "value": "444187722"
    }
  ]
},
"intervals": [
  "2015-11-09T02:00:00/2015-11-09T04:00:00"
]
}
```

在上述的查询语句中，我们通过 `granularity` 设置了查询粒度为 60000 毫秒（即 10 分钟）。如果将该查询作用到聚合粒度为 1 分钟的 `DataSource druid_metric`，都可以得到如下相同的结果（均仅取查询结果返回集的最后 一个结果做展示）：

```
.....
}, {
  "version": "v1",
```



```
"timestamp": "2015-11-09T03:50:00.000Z",
"event": {
  "metricId": "444187722",
  "applicationId": "1007323",
  "num1": 212
}
```

这就证明了，聚合粒度为 1 分钟的 DataSource druid_metric 在功能上不仅能够满足粒度为 1 分钟的聚合查询，还完全能够满足大于其最低粒度的 10 分钟的聚合查询并能返回正确的结果。但是，可能有读者又会好奇：这类查询的性能又如何呢？我们在实际数据上做过对比试验：分别在最低聚合粒度为 1 分钟、10 分钟和 1 小时的 DataSource 上做不同时间跨度与聚合粒度的查询。以下是其中的一组对比结果（为排除缓存的作用，仅记录了查询被首次执行所花费的时间）。

按 10 分钟聚合的对比数据如下表所示。

聚合粒度	按 10 分钟聚合	按 10 分钟聚合
DataSource	druid_metric	druid_metric_10minute
简单的 groupBy 查询	0m0.389s	0m0.194s

按 1 小时聚合的对比数据如下表所示。

聚合粒度	按 1 小时聚合	按 1 小时聚合
DataSource	druid_metric	druid_metric_1hour
简单的 groupBy 查询	0m0.298s	0m0.296s

按 24 小时聚合的对比数据如下表所示。

聚合粒度	按 24 小时聚合	按 24 小时聚合
DataSource	druid_metric_1hour	druid_metric
简单的 groupBy 查询	0m16.017s	0m0.577s

从上面的表中可以看出，在聚合查询的时间跨度即聚合的数据量不大的情况下，比如按 10 分钟聚合，虽然在不同的 DataSource 上查询执行时间的绝对时间相差也不大，但是依然基本符合一个规律：DataSource 本身的聚合粒度越接近聚合查询指定的粒度，查询的速度越快。这个规律在聚合的数据量变大的情况下，比如按 24 小时聚合时，这种性能的差距会被放大，从而使得差距变得很明显。正是因为这个原因，我们看到了金字塔方案带来的第一个优势：提供不同聚合粒度的 DataSource，让查询作用在更接近其粒度的 DataSource 上，从而能获得更佳的查询效率。

然而，查询性能的优势并不是我们选择金字塔方案的唯一原因，它有时能够让我们在存储空间方面获得更高性价比的存储效率，其实这是影响我们做出选择的一个更大因素。为了更好地理解这个优势点，我们先看一个根据自己的数据所做的对比表格（单位：GB，Segment Replication：1）。

DataSource	小时存储成本	天存储成本	月存储成本（31 天）
druid_metric	1.30	31.20	967.20
druid_metric_10minute	0.52	12.48	386.88
druid_metric_1hour	0.18	4.32	133.92

根据上表我们可以很轻松地得出一个符合推理的试验结论：聚合粒度越低时，单位时间所需的存储空间越大。与此同时，对于大多数时序数据库的使用场景来说，一般对于越近的时间范围内的查询所需要的聚合粒度越低，而对于越久远的时间范围内的查询所需要的聚合粒度越高，因为此时往往只需要查一个趋势而非细节。就拿 OneAPM 产品所提供的指标数据查询接口来说，对于最近几个小时，它能够提供更 1 分钟级别粒度的查询；对于最近几天，它能够提供更 10 分钟级别粒度的查询；而对于更久远的时间范围，比如 15 天甚至几个月，它就只会提供小时或更高粒度级别的查询了——这其实符合客户实际的查询需求。由于这个原因，金字塔方案不仅能够满足 OneAPM 产品对不同时间范围内的查询粒度需求，还能够在此基础上提供很高的存储性价比，因此很自然地被接纳而作为了数据存储的指导方案。

我们可以通过图 10-9 对 Druid 金字塔存储方案的特点做一个总结。

目前，OneAPM 不仅在其 SaaS 平台上通过使用 Druid 技术成功地完成了对其海量指标数据的实时消费与查询，也在其企业级产品中同样通过 Druid 处理了日益增长的企业数据，因此 Druid 已经成为了 OneAPM 技术栈中重要的一环。

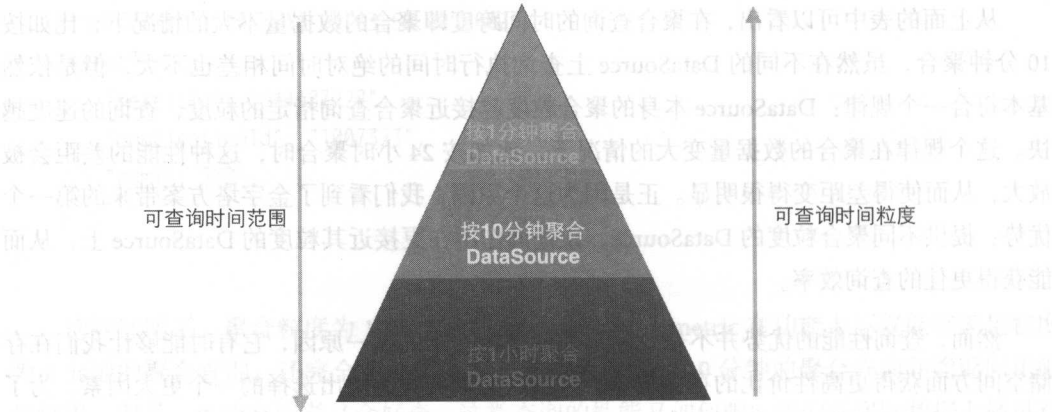


图 10-9 Druid 金字塔数据存储方案

按1分钟聚合 DataSource	按10分钟聚合 DataSource	按1小时聚合 DataSource	按1天聚合 DataSource
-------------------	--------------------	-------------------	------------------

10.5 小结

通过对上述不同企业技术团队对 Druid 的实践案例与经验的介绍，相信读者朋友们已经对 Druid 有了更加全面和深入的了解，甚至可能已经产生了一些比较具体的将 Druid 应用到自己公司解决方案或应用系统中的思路或方案。“纸上得来终觉浅，绝知此事要躬行”，如同学习其他技术一样，掌握 Druid 最好的方法就是实践，因此大家在对 Druid 有了一定的认识后应该尽快上手练习，并且争取早日将其应用到自己的实际工作中，在战斗中学习战斗，让在实践中碰到的问题驱动自己对 Druid 技术的学习和理解。通过实践的方式，相信大家不仅可以早日掌握 Druid 这项技术，让它为自己产生效益，并且还能够创新出其他的使用技巧与应用场景，从而为 Druid 技术的更广泛应用贡献自己的力量。

第11章

Druid 生态与展望

在当今软件行业中，一款软件的成功不仅需要依赖于自身生生不息的进化与完善，还常常需要仰仗其周边生态体系的繁荣与成熟。作为一款越来越受关注的软件产品，Druid 本身并未停止进化升级，其生态正在逐步发展壮大——这一切都彰显了 Druid 旺盛的生命力。

11.1 Druid 生态系统

最近几年数据分析软件层出不穷，Druid 在竞争激烈的行业中持续保持高速发展，其中重要原因就是 Druid 的生态系统日渐完整，包括数据管理能力、数据接口访问能力，甚至包括多个以 Druid 为核心的数据分析平台。另外，Druid 在互联网公司的广泛应用也不断推动着生态系统的完善。

如图 11-1 所示，从数据源、数据管理、数据接口、可视化等方面列举了一些相关的 Druid 生态软件，其中包括 Imply.io 公司及其产品，它的核心创始人都来自于 Druid 的核心开发人员。

由于 Druid 相关的开源项目比较多，很难在本章中对它们都进行详细的介绍，因此仅重点介绍 Imply.io 公司推出的几款组件（Plywood、PlyQL 和 Pivot）以及其他几个项目，以供读者了解学习和使用这类开源组件的基本思路和方法。本章只会对“Druid 数据分析生态系统”图中提及的一些主要组件做简单的说明。

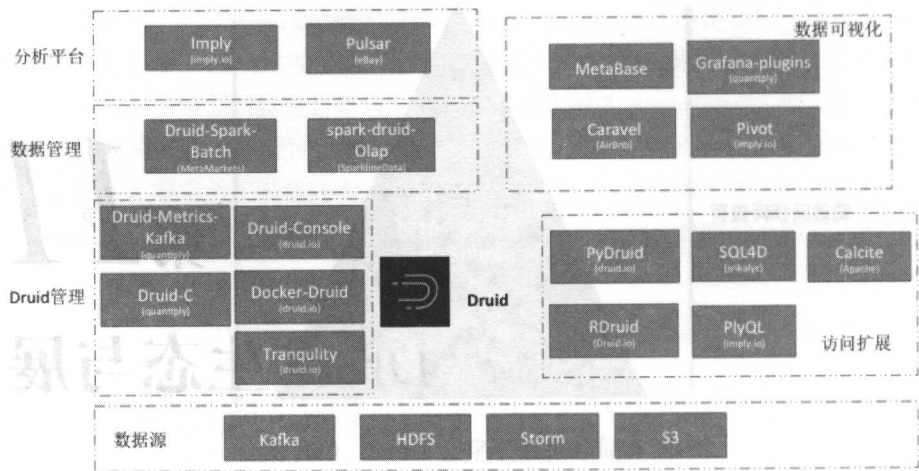


图 11-1 Druid 数据分析生态系统

1. 数据源

- Kafka: 分布式消息订阅发布系统，具备很强的容错性和可扩展性，已经成为了大数据相关领域事实上的标准组件。网址：<http://kafka.apache.org>。
- HDFS: 使用极其广泛的分布式文件，属于大名鼎鼎的 Hadoop 项目的一部分。网址：<http://hadoop.apache.org>。
- Storm: Twitter 开源的分布式实时计算系统，可以轻松读取不同数据源的数据，并将计算结果输出到不同的数据源或计算系统。网址：<http://storm.apache.org>。
- S3: Amazon Simple Storage Service (Amazon S3) 为开发人员和 IT 团队提供安全、耐久且扩展性高的云存储。网址：<https://aws.amazon.com/cn/s3>。

2. Druid 管理

- Druid-Metrics-Kafka: 帮助用户收集并转发 Druid 指标数据到 Kafka 上，省去了自己搭建 HTTP Server 和写 Kafka Producer 的相关工作。网址：<https://github.com/quantiply/druid-metrics-to-kafka>。
- Druid-Console: Druid 项目中 Coordinator 组件自身所带的交互界面，用户可以通过该界面访问 DataSource 和 Segment 等相关信息。网址：<http://druid.io/docs/latest/design/coordinator.html>。
- Docker-Druid: 帮助用户通过 Docker 迅速搭建起 Druid 集群。网址：<https://github.com/>

druid-io/docker-druid。

- Tranquility: Tranquility 帮助用户发送实时数据流到 Druid, 并且能够操作数据文件的分区、复制等任务, 使得用户能够灵活地完成 Druid 数据消费任务。网址: <https://github.com/druid-io/tranquility>。

3. 访问扩展

- PyDruid: 提供了访问 Druid 的 Python 接口。网址: <https://github.com/druid-io/pydruid>。
- SQL4D: 提供了访问 Druid 的 SQL 接口。网址: <https://github.com/srikalyc/Sql4D>。
- RDruid: 提供了访问 Druid 的 R 接口。网址: <https://github.com/druid-io/RDruid>。
- PlyQL: 基于 Plywood 项目, 提供了访问 Druid 的 SQL 接口。网址: <https://github.com/implydata/plyql>。
- Calcite: Apache Calcite 是分布式系统领域里的一个查询引擎。它能够连接各种数据源, 并提供了标准的 SQL 语言、多种查询优化、OLAP 和流处理等接口。Calcite 提供了 Druid 的适配器, 使得用户可以同时通过 Calcite 查询 Druid 和其他数据源的数据, 并做综合处理。网址: <https://calcite.apache.org>。

4. 数据管理

- Druid-Spark-Batch: 使用户能够通过 Spark 批处理任务完成 Druid 的数据索引工作。网址: <https://github.com/metamx/druid-spark-batch>。
- Spark-Druid-OLAP: Sparkline BI Accelerator 是基于 Spark SQL API 实现的一个商务智能 (Business Intelligence, BI) 计算引擎。它能够在逻辑立方体或星型模型上快速执行各类查询, 降低了企业在大数据技术栈 (Hadoop/Spark) 上进行随机查询的难度。网址: <https://github.com/SparklineData/spark-druid-olap>。

5. 分析平台

- Imply: 由 Druid 主要创始人创建的公司, 其提供的 IAP 平台包含了若干个基于 Druid 的开源项目。网址: <https://github.com/implydata>。
- Pulsar: 由 eBay 开源的实时分析平台和流处理框架, 具有高扩展性、高可用性和基于事件驱动等特征, 能够实时收集与处理用户行为和业务事件。它提供了一种 SQL 的事件处理语言, 并且用户可以基于它自定义流事件, 并最终完成对数据的加工 (聚合、补充、变异和过滤等)。网址: <https://github.com/pulsarIO>。

6. 数据可视化

- MetaBase: 帮助企业用户快捷实现商务智能与分析的软件, 它提供了连接多个数据源的接口, 如 Druid、MySQL、PostgreSQL、Oracle、SQL Server、MongoDB、SQLite 与 Google BigQuery 等。网址: <https://github.com/metabase>。
- Grafana-Plugins: 作为优秀的前端展示组件, Grafana 通过 Plugins 扩展了其功能。通过对 Druid 的支持插件, Grafana 实现了对 Druid 的访问功能。网址: <https://github.com/grafana/grafana-plugins/tree/master/datasources/druid>。
- Caravel: 来自 AirBnB 公司, 与 Pivot 一样是一个开源的数据可视化平台 (之前的名字叫作 Panoramix)。网址: <https://github.com/airbnb/caravel>。
- Pivot: 基于 Plywood 实现的对 Druid 进行交互式数据探索的开源组件。与 Plywood、PlyQL 一样由 Imply 开源。网址: <https://github.com/implifydata/pivot>。

11.2 Druid 生态系统资源

目前 Druid 生态系统中的开源项目主要分三类。

- druid.io 在 github 上的项目 (地址: <https://github.com/druid-io>), 比如 Druid 核心项目、Tranquility、Docker-Druid、Druid-Benchmark、Druid-Console、RDruid 和 PyDruid 等都是 Druid 官方提供的项目, 这些项目的积极贡献者则有可能成为 druid.io 社区的 Committer 甚至 PMC 成员。
- Imply 公司在 github 上的项目 (地址: <https://github.com/implifydata>), 比如 Plywood、PlyQL、Pivot、plywood-mysql-requester 和 plywood-proxy 等项目, 属于 Druid 创始人建立的 Imply.io 所发展的开源项目。
- 其他 Druid 相关开源项目, 比如 Druid-Metrics-Kafka (地址: <https://github.com/quantiply/druid-metrics-to-kafka>) 等。

11.2.1 IAP

Imply 公司是 Druid 创始人创建的公司, 旨在提供主要基于 Druid 的技术服务, 并成为推动 Druid 这个开源项目持续发展的中坚力量。目前该公司也基于稳定的 Druid 版本提供了几个开源组件, 以帮助用户更好地基于 Druid 搭建自己的应用, 而这几个开源组件加上 Druid 便形成了 IAP (Imply Analytics Platform), 如图 11-2 所示。

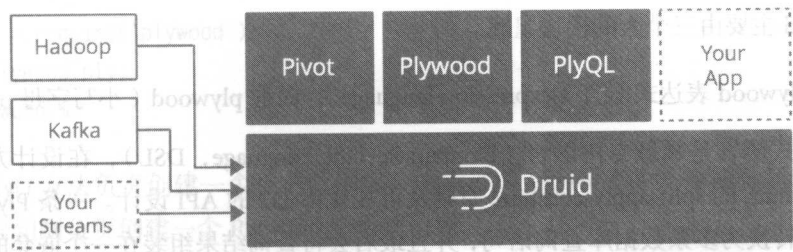


图 11-2 IAP

IAP 将 Druid 集群的所有节点分为三类，如图 11-3 所示。

- 查询服务器（Query Server），包含 Druid 查询节点、Pivot 和 PlyQL。
- 数据服务器（Data Server），包含 Druid 历史节点和中间管理者。
- 控制服务器（Master Server），包含 Druid 协调节点和统治节点。

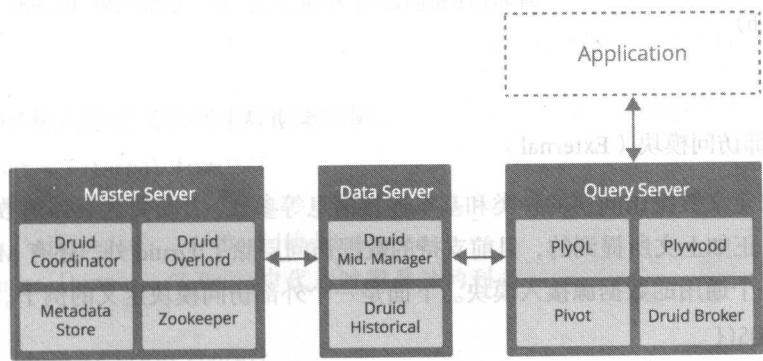


图 11-3 IAP 集群节点分类

目前 IAP 主要提供的新开源组件有三个，即 Plywood、PlyQL 和 Pivot。它们从技术上进一步提高了 Druid 的可访问性，大大方便了用户对 Druid 的使用，因此 IAP 也将它们列为查询服务器一类。

11.2.2 Plywood

Plywood 是介于数据存储层与数据访问层之间的一个代理层，旨在协助用户更容易地访问数据存储层数据及创建基于数据的应用。Plywood 本质上是一个 JavaScript 库，可运行在 Node.js 运行环境中。值得一提的是，Plywood 的数据存储层不局限于支持 Druid 数据源，目前还支持 MySQL，并且以后会支持更多的数据源。

Plywood 主要由三个大的模块组成。

(1) Plywood 表达式语言 (expression language), 简称 plywood (小写字母 p 开头)

该表达式语言是领域专用语言 (Domain Specific Language, DSL), 在设计方面借鉴了 Hadley Wickham 的 split-apply-combine 原则及可视化库 D3 的 API 设计。一条 Plywood 原生语句能够被转换为多条数据库查询语句, 并且最后会将查询结果组装在一个嵌套的数据结构中返回, 这样无疑会大大降低 Druid 数据访问层的复杂度。下面是一个 Plywood 原语的例子。

```
var ex = ply()  
  .apply('Count', $('wiki').count())  
  .apply('TotalAdded', '$wiki.sum($added)')  
  .apply('Pages',  
    $('wiki').split('$page', 'Page')  
    .apply('Count', $('wiki').count())  
    .sort('$Count', 'descending')  
    .limit(6)  
  );
```

(2) 外部访问模块 (External)

主要用来定义数据访问层的种类和基本连接信息等参数, 并负责为具体的数据源生成查询执行计划。正如上文所提到的, 目前支持的数据访问层除了 Druid 外, 还有 MySQL。总而言之, 这是一个通用的数据源接入模块。下面是一个外部访问模块定义的例子。

```
External.fromJS({  
  engine: 'druid',  
  dataSource: 'wikipedia', // Druid中DataSource的名字  
  timeAttribute: 'time', // Druid中代表时间属性的列名  
  requester: druidRequester // 负责完成Druid请求的实例  
})
```

(3) 帮助模块 (Helper)

为方便使用 Plywood 提供了一些帮助函数。

可以通过 npm 轻松地完成 Plywood 的安装: “npm install plywood”, 也可以在浏览器中使用 Plywood。现在为大家举一个 Imply.io 官网上的例子, 以便对 Plywood 有更感性的认识。一般来说, 使用 Plywood 分为以下几个步骤。

首先, 像其他产品或项目一样, 在使用某个库或其依赖之前需要先将它们引入到程序中。

```
var plywood = require('plywood');  
var ply = plywood.ply;  
var $ = plywood.$;
```

其中 `ply()` 方法负责创建一个空的数据集，之后许多 Plywood 的操作都会基于该数据集发生；`$()` 语句则负责创建一个 Plywood 原语的引用。

在访问 Druid 数据源之前，需要先得到对应的 External 和 `druidRequesterFactory` 实例。

```
var External = plywood.External;  
var druidRequesterFactory = require('plywood-druid-requester').druidRequesterFactory;
```

接下来，便可以通过 `druidRequesterFactory` 实例得到访问 Druid 数据源所必需的 `druid-Requester`。

```
var druidRequester = druidRequesterFactory({  
  host: '192.168.60.100:8082' // 定义用户自己的Druid路径  
});
```

通过 JSON 格式的定义来创建数据集实例。

```
var wikiDataset = External.fromJS({  
  engine: 'druid',  
  dataSource: 'wikipedia', // Druid的dataSource  
  timeAttribute: 'time', // Druid中代表时间属性的列名  
  context: {  
    timeout: 10000 // Druid context  
  }  
}, druidRequester);
```

为了方便在执行环境中更好地定义查询语句，我们还可以创建执行上下文（`context`），并在其中自定义变量等。

```
var context = {  
  wiki: wikiDataset,  
  seventy: 70  
};
```

此时，程序便可以执行具体的查询语句了。

```
var ex = ply()  
// 指定External，以及作用于该External的基于时间和语言的过滤器
```

```

.apply("wiki",
  $('wiki').filter($('time').in({
    start: new Date("2015-09-12T00:00:00Z"),
    end: new Date("2015-09-13T00:00:00Z")
  })).and($('channel').is('en')))
)

// 计算数量
.apply('Count', $('wiki').count())

// 计算“added”属性列的总和
.apply('TotalAdded', '$wiki.sum($added)');

// 引用在context中定义的变量seventy
.apply('70', $('seventy'));

ex.compute(context).then(function(data) {
  // 将数据转换成可读的格式并记录到日志中
  console.log(JSON.stringify(data.toJS(), null, 2));
}).done();

```

得到查询结果。本例的查询结果是只包含一条数据的数据集。

```

[
  {
    "70": 70,
    "TotalAdded": 32553107,
    "Count": 113240
  }
]

```

鉴于 Plywood 的设计理念和工程特性，通常使用 Plywood 访问数据存储层有多种方法。

- 用户创建基于网络和数据驱动的应用，并且后台服务运行于 Node.js 运行环境中，这时可以在网络应用层和 Node.js 运行环境中均使用 Plywood 作为数据访问代理。此时，网络应用层中的 Plywood 组件可以向 Node.js 运行环境中的 Plywood 组件发送基于 Plywood 原语的查询请求，并随后接收来自后者的返回结果；Node.js 运行环境中的 Plywood 组件则会接收到的 Plywood 原语的查询请求翻译成一系列数据访问层的查

询语言，并对结果进行封装，如图 11-4 所示。本章后面介绍的 Pivot 组件便是采用这种架构。

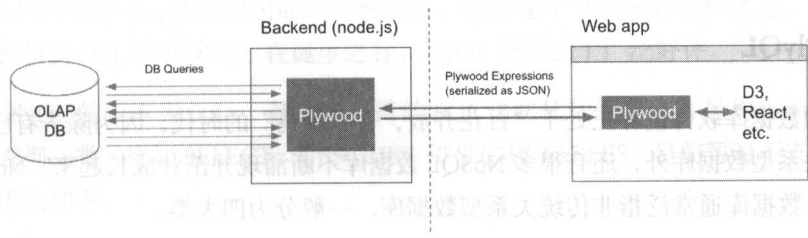


图 11-4 基于 Plywood 和使用 Node.js 的网络应用架构示意图

- 应用并没有使用 Node.js 运行环境，而是在网络应用层直接通过 Plywood 组件访问数据存储层，如图 11-5 所示。

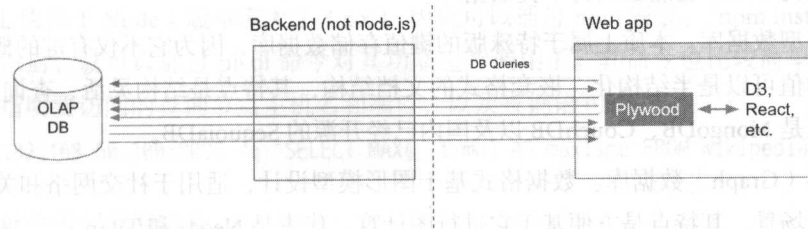


图 11-5 直接通过 Plywood 组件访问数据存储层

- 当然，在很多场合中，在网络应用层直接通过 Plywood 访问数据存储层并不是符合最佳实践的模式，起码不符合现代系统分层的主流架构思想，但又不是所有应用的后端都会运行在 Node.js 运行环境中，因此这就形成了既基于两层 Plywood 组件又不依赖于 Node.js 的架构模式，如图 11-6 所示。本章后面介绍的服务模式的 PlyQL 组件便是采用这种架构。

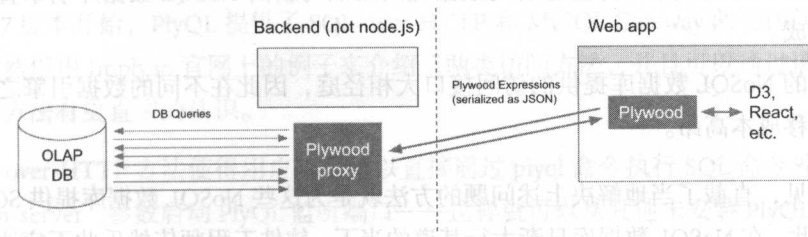


图 11-6 既基于两层 Plywood 组件又不依赖于 Node.js 的架构示意图

以上都是对使用 Plywood 组件的不同架构的介绍,用户可以根据自己的现实需求来决定使用何种方式通过 Plywood 组件简化 Druid 数据访问应用。

11.2.3 PlyQL

当今的数据库软件世界正处于“百花齐放,百家争鸣”的时代,因为除了已经有非常成熟的传统关系型数据库外,还有很多 NoSQL 数据库不断涌现并茁壮成长起来。NoSQL (Not Only SQL) 数据库通常泛指非传统关系型数据库,一般分为四大类。

- 键值 (Key-Value) 存储数据库。通过键值对的方式组织数据格式,其特点是上手简单,部署容易。代表是 Redis。
- 列存储数据库。数据行中仍有键的概念,但是各个属性列被分别存放在不同的数据文件中,其特点是适合在分布式系统中处理海量数据。代表是 HBase 和 Cassandra。笔者认为 Druid 也能被归为本类数据库。
- 文档型数据库。本质上属于特殊版的键值存储数据库,因为它不仅有键的概念,而且它的值可以是半结构化、嵌套格式的文档结构,其特点是结构灵活,查询效率较高。代表是 MongoDB、CouchDB 以及国内已经开源的 SequoiaDB。
- 图形 (Graph) 数据库。数据格式基于图形模型设计,适用于社交网络和关系图谱等应用场景,其特点是方便基于它进行图计算。代表是 Neo4J 和 Titan。

时至今日, NoSQL 数据库早已不是停留在论文里的纯概念或“黑科技”,而是在许多解决方案中都已经占有一席之地,特别是在大数据处理领域、对事务等传统数据库特性要求不高的场景中。目前 NoSQL 数据库已经成为各类解决方案的首选,并且以出色的表现证明了自己的价值,因此在这些解决方案中基本上替代了传统数据库。然而,即使 NoSQL 数据库作为引擎已经得到了广泛接受,但是从它们所提供的访问接口来看依然存在诸多问题。

- NoSQL 数据库提供的访问接口不同于传统的 SQL,因此软件工程师学习 NoSQL 数据库需要额外的成本,并且已有的数据应用难以与新的 NoSQL 数据库引擎直接进行无缝集成。
- 不同的 NoSQL 数据库提供的访问接口大相径庭,因此在不同的数据引擎之间进行数据迁移成本高昂。

显而易见,直截了当地解决上述问题的方法就是为这些 NoSQL 数据库提供 SQL 访问接口。正因如此,在 NoSQL 数据库日渐大行其道的当下,软件工程师依然乐此不疲地为不同的 NoSQL 数据库编写类似的 SQL 访问接口,比如基于 Hadoop HDFS 文件系统和 MapReduce 计

算框架的 Hive 组件，以及基于 HBase 的 Phoenix 项目等。对于同样是 NoSQL 数据库的 Druid 来说，为其提供一个类 SQL 访问接口同样具有积极意义，所以 PlyQL 组件因此产生了。

PlyQL 同样属于 Impley 公司出品的开源组件，它搭建于 Plywood 组件之上，并在其命令行工具中提供类 SQL 访问接口。在诞生之日，PlyQL 便受到了广泛好评。

目前 PlyQL 的版本为 0.8.x，其主要支持的 SQL 语法比较简单，并且均属于纯粹将数据从库往外查询一类，比如 SELECT、DESCRIBE 和 SHOW TABLES。目前暂时不支持但正在开发当中的语法如下。

- 查询仿真。旨在预览即将被执行的查询，而非真正执行它们。
- 在 WHERE 子句中提供子查询功能。
- 对 JOIN 算子的支持。
- 窗口函数。

PlyQL 依赖于 Node（版本需大于 4.x.x），因此可以通过 npm 安装：“npm install -g plyql”。安装完成以后，就可以通过 plyql 命令对其功能进行调用了。其命令也比较简单，一般需要在命令中指明要访问的查询节点主机名和端口，以及查询语句。

```
plyql -h 192.168.60.100:8082 -q 'SELECT MAX(__time) AS maxTime FROM wikipedia'
```

然后就会得到返回结果。

```
[
  {
    "maxTime": {
      "type": "TIME",
      "value": "2015-09-12T23:59:00.000Z"
    }
  }
]
```

从 0.7 版本开始，PlyQL 提供了 SQL-over-HTTP 和 MySQL Gateway 两类访问方法。接下来我们依然借用 Impley.io 官网上的例子来介绍这两类访问方法，并且可以通过该介绍让大家对其使用方法有更直观的认识。

SQL-over-HTTP 方法使得用户除了可以直接通过 plyql 命令执行 SQL 命令外，还可以通过 “--json-server” 参数启动 PlyQL 监听端口——这样就可以从其他未安装 PlyQL 的机器上通过网络访问该监听端口来使用 PlyQL 所支持的 SQL 语法了。以下是启动 PlyQL 监听端口的命令实例。

```
plyql -h your.druid.broker:8082 -i P2Y --json-server 8083
```

然后就可以通过 `curl` 命令向该监听端口发送查询命令。

```
curl -X POST 'http://localhost:8083/plyql' \  
-H 'content-type: application/json' \  
-d '{  
  "sql": "SELECT COUNT(*) FROM wikipedia WHERE \"2015-09-12T01\" <= __time AND __time <  
    \"2015-09-12T02\""}'  
}'
```

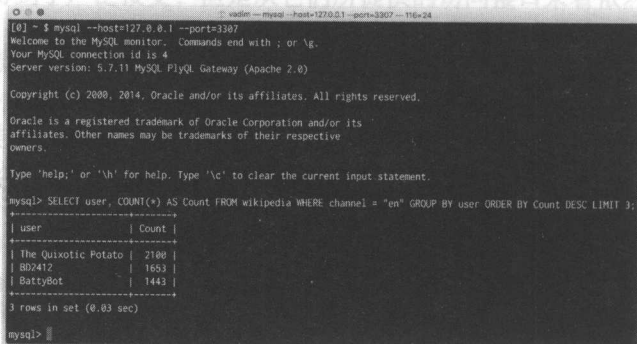
不难看出，SQL-over-HTTP 方法调用让 PlyQL 的使用变得更加容易。

现在我们再来看 MySQL Gateway 访问方法。虽然 SQL-over-HTTP 方法提供了 HTTP 的访问方法，这符合大多数 NoSQL 数据库的访问风格设计，也与 Druid 原生提供的访问方法一脉相承，但是它却不算是符合传统关系型数据库的经典访问方式。对于传统关系型数据库，一般会通过 JDBC/ODBC 驱动或基于其建立的客户端对后台数据库数据进行访问，这使得许多现存的数据应用已基本通过这类方式来访问数据库，而且其使用者都已经熟悉了该方法。与此同时，MySQL 作为目前最成功的开源传统关系型数据库，它拥有大量的应用支持和用户基础。因此，为了与已有的 MySQL 数据库应用和访问客户端进行集成，PlyQL 提供了 MySQL Gateway 访问方法。

要使用 MySQL Gateway 访问方法，首先也需要开启相关的监听端口（通过“`--experimental-mysql-gateway`”参数）。

```
plyql -h your.druid.broker:8082 -i P2Y --experimental-mysql-gateway 3307
```

然后，用户便可以通过 MySQL 客户端对 Druid 数据进行访问了，如图 11-7 所示。



```
mysql --host=127.0.0.1 --port=3307  
Welcome to the MySQL monitor.  Commands end with ; or \g.  
Your MySQL connection id is 4  
Server version: 5.7.11 MySQL PlyQL Gateway (Apache 2.0)  
  
Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.  
  
Oracle is a registered trademark of Oracle Corporation and/or its  
affiliates. Other names may be trademarks of their respective  
owners.  
  
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.  
mysql> SELECT user, COUNT(*) AS Count FROM wikipedia WHERE channel = "en" GROUP BY user ORDER BY Count DESC LIMIT 3;  
+-----+-----+  
| user          | Count |  
+-----+-----+  
| The Quixotic Potato | 2180  |  
| BD2412        | 1693  |  
| BattyBot      | 1443  |  
+-----+-----+  
3 rows in set (0.03 sec)  
  
mysql>
```

图 11-7 通过 MySQL 客户端访问 Druid 数据

除了 MySQL 客户端, 用户也可以通过 Java 程序使用 JDBC 驱动来访问 PlyQL Gateway 以查询 Druid 数据。Imple.io 提供的实例程序如下:

```
import java.sql.SQLException;
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;

class DruidQuery
{
    public static void main(String[] args) throws SQLException
    {
        Connection con = DriverManager.getConnection("jdbc:mysql://127.0.0.1:3307/plyql1");
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(
            "SELECT page, count(*) AS cnt FROM wikipedia GROUP BY page ORDER BY cnt DESC LIMIT
            15"
        );
        while (rs.next()) {
            String page = rs.getString("page");
            long count = rs.getLong("cnt");
            System.out.println(String.format("page[%s] count[%d]", page, count));
        }
    }
}
```

综上所述, PlyQL 不仅提供了类 SQL 的查询语法, 而且提供了丰富的访问方法, 这无疑大大便利了 Druid 的 SQL 调用解决方案的实施。

11.2.4 Pivot

人类习惯通过视觉来获取信息, 因为大脑超过 80% 的信息都是通过眼睛获得的。俗话说“一图胜千言”, 正道明了图形在信息表达方面的突出优势。在计算机数据处理软件领域, 虽然绝大多数有价值的内容都是结构化的数值型数据, 但是人们依然喜欢通过图表之类的可视化途径来感知、操控数据, 因此许多数据分析软件的成功常常得益于其出色的数据可视化模块。这个规律在 Druid 所处的时序数据处理软件领域依然生效, 因为通过数据可视化模

块，人们可以很容易地理解数据所拥有的规律和特点，也能较轻松地发现存在于数据中的异常元素等重要信息。进一步而言，通过与数据互动式的分析操作，人们能够更加具有创造性和动态地解构与分析数据，以及完成对不同数据进行多维度的比较分析等操作，对数据的分析手段得到了极大的增强，有力地深化了用户对数据价值的挖掘。面对数据可视化方面的需求，Impley 推出了 Pivot 这款开源数据可视化产品。

Pivot 基于 Plywood 组件创建，是一款基于网页的 Druid 数据可视化探索产品，它能够方便用户基于 Druid 数据进行 OLAP 分析，并且能够将数据以可视化的方式进行展示。

与上述的 Plywood 和 PlyQL 一样，Pivot 的安装也十分方便——直接通过以下命令安装即可。

```
npm i -g imply-pivot
```

安装完成以后，还需要将 Pivot 和 Druid 集群中接收查询请求的查询节点联系起来，命令如下：

```
pivot --druid your.druid.broker.host:8082
```

值得注意的是，即使完成了联系查询节点的操作，Pivot 也依然有可能无法识别一些集群配置，从而导致启动失败。为了解决这个问题，可以通过“--config”参数来手工指定 config 文件，让 Pivot 获得正确的集群配置。命令如下：

```
pivot --config config.yaml
```

关于 config.yaml 的配置语法，可具体参考官方文档 <https://github.com/implydata/pivot/blob/master/docs/configuration.md>，在此不再赘述。

如果想知道 Pivot 默认获得的配置信息，可以使用以下命令：

```
pivot --druid druid.broker.host:8082 --print-config --with-comments > config.yaml
```

Pivot 正确识别出 Druid 集群配置后，就可以通过浏览器使用 Pivot 组件了。Pivot 的 UI 设计比较简洁、易懂，也支持提供基于页面的拖拽（drag-and-drop）操作，因此用户基本上上手就直接使用。归结起来，用户的操作可以分别基于维度和 Measure。对于 Measure，操作比较简单，主要选择展示哪些 Measure 的值即可；而对于维度，可以在以下两个方面对查询进行定义。

- **Filter**：定义通过哪些具体的维度值对结果集进行过滤。实际上就相当于查询条件中的 WHERE 子句的作用。

- Split: 定义通过哪些维度对结果进行分组。实际上就相当于查询条件中的 GROUP BY 子句的作用。

下面列举几个实际的图例作为演示说明。

图 11-8 展现的是一个对多个 Measure 仅通过时间维度进行 Split 操作的示例，并且在该示例中还通过在 Filter 中指定“Namespace 值为 template”这一条件对结果集进行了过滤。

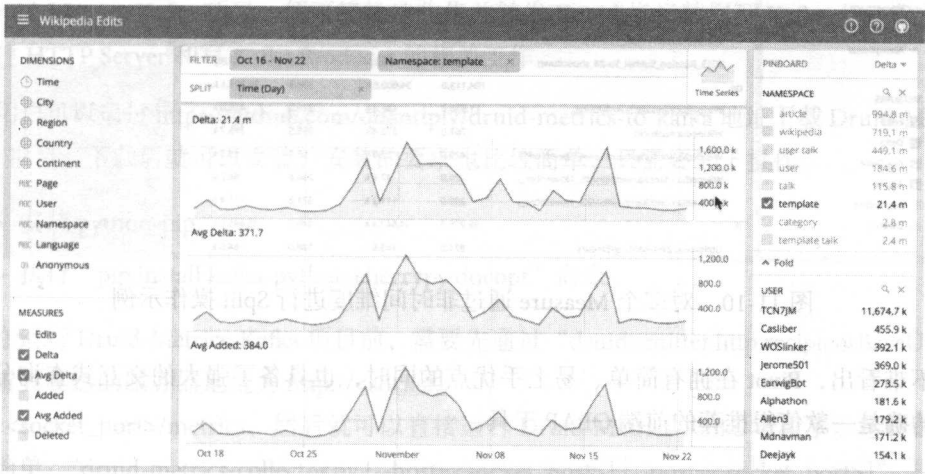


图 11-8 对多个 Measure 仅通过时间维度进行 Split 操作示例

图 11-9 展现的是一个仅对单个 Measure 通过时间和其他维度进行 Split 操作的示例。

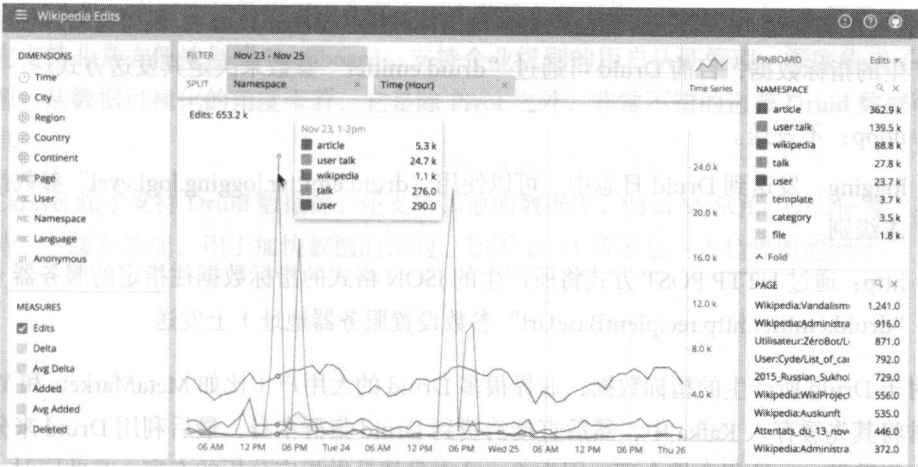


图 11-9 对单个 Measure 通过时间和其他维度进行 Split 操作示例

图 11-10 展现的是一个对多个 Measure 通过非时间维度进行 Split 操作的示例。

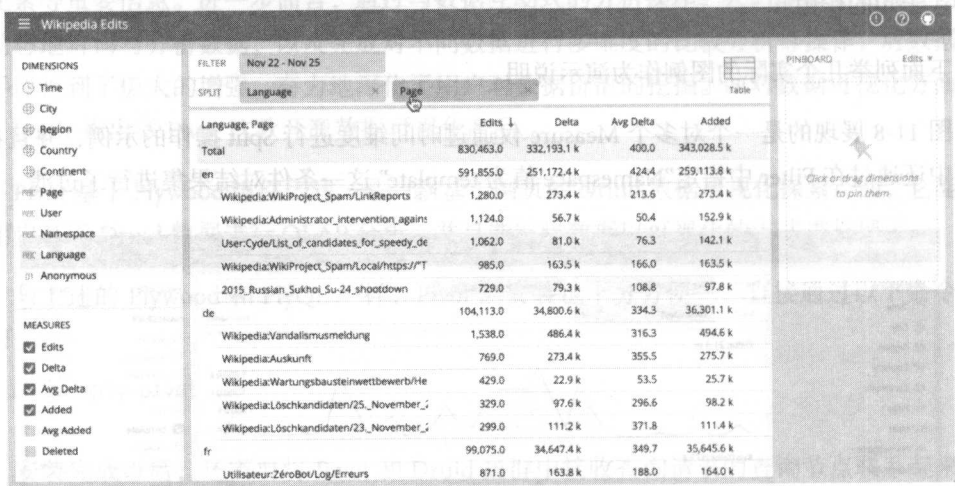


图 11-10 对多个 Measure 通过非时间维度进行 Split 操作示例

不难看出，Pivot 在拥有简单、易上手优点的同时，也具备了强大的交互式查询分析功能，的确是一款值得推荐的前端 OLAP 工具。

11.2.5 Druid-Metrics-Kafka

对于非 druid.io 和 Imply 公司主导的 Druid 相关开源软件，我们拿 Druid-Metrics-Kafka 项目来做介绍。

Druid 的各个服务在运行时都会产生多种指标数据（Metric Data），供监控管理使用。对于所产生的指标数据，目前 Druid 可通过“druid.emitter”参数来决定其发送方式。

- noop：不发送。
- logging：发送到 Druid 日志中。可以使用“druid.emitter.logging.logLevel”参数指定日志级别。
- http：通过 HTTP POST 方式将所产生的 JSON 格式的指标数据往指定的服务器（通过“druid.emitter.http.recipientBaseUrl”参数设置服务器地址）上发送。

对于 Druid 所产生的指标数据，业界很多 Druid 的大用户（比如 MetaMarkets 和 Yahoo）都常常将其先缓存入 Kafka 中，然后直接消费到 Druid 集群本身，最后利用 Druid 来分析自己的指标数据——这很合情合理，因为 Druid 本身就是做相应分析的专家。与此同时，很多用户喜欢使用 HTTP 方式让 Druid 往外发送指标数据，并用 HTTP Server 来接收，因为这种

方式在分布式系统中比较方便，特别是在集群规模比较大的时候。

如果选择了上述方式，那么在数据最后进入 Druid 之前很明显有以下两个工作要完成。

- 创建并启动一个 HTTP Server 来接收 Druid 通过 HTTP 方式发送的指标数据。
- 将 HTTP Server 所接收的指标数据转发到 Kafka 上。

开源项目 Druid-Metrics-Kafka 的目标正是为了完成这两个工作，所以用户只要安装使用了 Druid-Metrics-Kafka 项目，便可轻松地收集并转发 Druid 指标数据到 Kafka 上，省去了自己搭建 HTTP Server 和写 Kafka Producer 的相关工作。

用户可以通过 <https://github.com/quantiply/druid-metrics-to-kafka> 地址下载 Druid-Metrics-Kafka 项目。下载后就可以安装，安装的要求也比较简单，只需要如下操作。

- 安装 python-pip。
- 执行 “pip install kafka-python cherrippy docopt” 命令。

在运行 Druid-Metrics-Kafka 项目前，需要先通过 “druid.emitter.http.recipientBaseUrl” 参数设置其发送的服务器地址为 `http://<socket_host>:<socket_port>/metrics`，然后就可以直接运行 Druid-Metrics-Kafka 项目了。运行的命令也很简单：“druid-metrics-collector.py [--host=<socket_host>] [--port=<socket_port>]”。运行起来后，所有的 Druid 指标数据都会被转发到指定的 Kafka Topic。

11.2.6 Caravel (Airbnb)

Caravel 是一个开源的数据可视化平台（之前的名字叫作 Panoramix），来源于 Airbnb 公司。它支持非常方便地创建 Dashboard，支持企业级别的用户认证管理，深度集成了 Druid 数据源。从数据可视化的角度来看，它是除 Pivot 之外，非常不错的选择 Druid 数据可视化的工具。

Caravel 除了支持 Druid 数据源，还支持其他的数据库，例如 MySQL、Oracle 等，其内部还支持一些缓存功能，用于加快数据的访问。如图 11-11 所示是一个趋势图的例子。

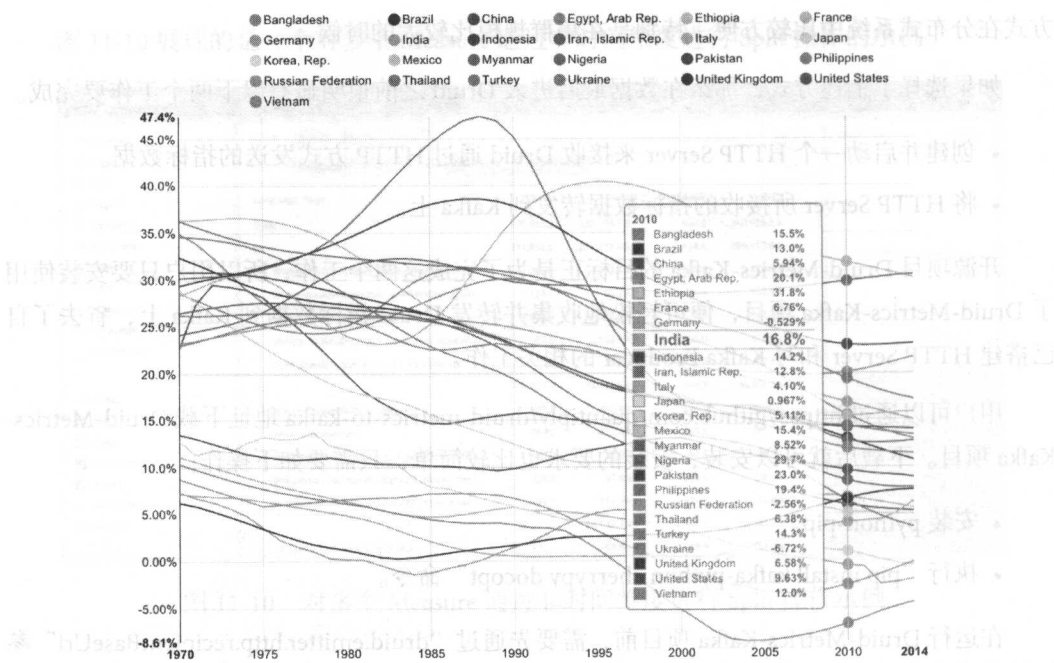


图 11-11 趋势图

11.3 Druid 的社区讨论组

值得一提的是，与其他成功的开源软件一样，Druid 的社区讨论组在 Druid 生态系统的建设中起到了十分积极的作用。目前 Druid 的社区讨论组主要有如下两个。

- 普通用户讨论组 (druid-user@googlegroups.com)：Druid 用户关于 Druid 使用相关问题的讨论组。可以通过 <https://groups.google.com/forum/#!forum/druid-user> 加入。
- 开发人员讨论组 (druid-development@googlegroups.com)：Druid 用户关于 Druid 开发相关问题的讨论组。可以通过 <https://groups.google.com/forum/#!forum/druid-development> 加入。

11.4 Druid 展望

在大数据开源组件的生态系统中，Druid 算是比较晚入场的一位——它在 2013 年年底才被开源出来，但是这并没成为它被大家广泛接受的一个阻碍。事实上，在短短不到三年的时间里，Druid 不但在国外有了如 Yahoo、MetaMarkets、Netflix 这类的大玩家做背书，在中国

国内也逐渐培养起来一大批主要来自于前沿互联网公司的拥趸，在发展速度上可谓日行千里——这无疑证明了 Druid 在技术方面的确有其独到并领先之处。然而，这仅仅是故事的开始，接下来 Druid 的发展很可能会更加迅猛，因为有一些关于 Druid 发展的利好因素正在日趋成熟。

- 用户社区的壮大。Druid 的创始团队正在努力将 Druid 推广到更强大的开源社区平台当中，以利用社区的力量来推广和发展 Druid。其中，大名鼎鼎的 Apache 社区便是一个极有可能的选项——一旦双方达成一致，那么对 Druid 来说就像完成了“招安”大计，对其今后的发展无疑起到了极大的推动作用，甚至有可能使其像目前功成名就的 Apache Hadoop 和 Apache Hive 等项目一样成为大数据领域某一方面事实上的标准。与此同时，中国国内的 Druid 社区也在不断发展壮大，并且定期举办很多线下的技术沙龙等活动，对培养国内用户起到了重要的积极作用。
- 商业公司的兴起。Druid 的几位创始人刚创建了 Imply 公司，旨在为客户提供基于 Druid 技术的商业服务。有了商业活动的支持，Druid 势必会得到更有力的公司级别的支持与推动（本章介绍的优秀的 Druid 相关组件 Plywood、PlyQL 和 Pivot 就是很好的例证），来自于商业客户的需求也将促进 Druid 自身向着更加完善的方向发展。
- 技术环境的优化。无论是“用户社区的壮大”还是“商业公司的兴起”，本质上都属于促进 Druid 健康发展的辅助因素，而真正能从根本上决定一项技术发展的因素永远都是实际需求，即技术发展所需的环境。当需求旺盛时，技术的发展便如烈火烹油一般；反之，当需求萎靡时，技术的前进动力便如同无本之木。幸运的是，目前 Druid 所处的技术环境正处于蓬勃发展的状态之中：互联网和物联网技术与生态依旧保持着迅猛发展的势头——它们在继续创造更加海量的时序数据的同时，也对海量数据做实时 MOLAP 分析有着愈发旺盛的需求，这一切无疑是 Druid 发展所需的必要土壤与养分。

根据上述的分析，我们有理由相信 Druid 项目不仅正在经历一个成功的今天，也必将拥有一个更加绚烂的明天。此时此刻，我们为这一切感到激动的原因，绝不仅仅是因为又多了一个能帮助我们解决某个技术领域问题的开源项目，更重要的是我们正在一起经历一个软件发展的美好时代——这是一个开源软件真正蓬勃发展的大好年代，也是一个属于愿在软件领域自由和民主协作的人们的黄金时代。

参考资料

<http://imply.io>

<https://github.com/implydata>

附录 A

常见问题（FAQ）

本附录列出了 Druid 的一些常见的问题，并针对这些问题做了简要的回答，希望能够帮助读者解决一些实际应用当中的问题。

A.1 写入

1. Druid 是否支持非结构化数据？

Druid 不支持非结构化数据，数据摄入时依赖预先定义好的结构对数据进行处理。

2. Druid 支持什么格式的数据入库？

Druid 支持 JSON、CSV、TSV 或者有明确分隔符的原始数据。除此之外，Druid 还支持自定义格式，需要自己编写 RegEx 或者 JavaScript 解析。

3. Druid 数据写入失败的原因？

- 实时：实时流数据的写入有一个以当前时间为基准的时间窗口（windowPeriod）设置，若待写入数据的时间不在该时间窗口内，则无法写入 Druid。
- 离线：离线写入的数据时间需在指定的 interval 范围内。

4. 实时写入方式的 Segment 没有落地的原因？

- Druid 更新 Metadata Storage 失败，需要确保连接 Metadata Storage 正确。

- 历史节点没有足够的容量用于本地缓存 Segment。
- Deep Storage 连接配置不正确。

5. 如何使用 HDFS 作为 Deep Storage?

- 确保安装包中包含 druid-hdfs-storage 组件。
- 在 common.runtime.properties 的 druid.extensions.loadList 配置中加入 druid-hdfs-storage, 且在 middleManager 配置中加入对应的 Hadoop 依赖项。
- 在服务启动命令中增加 Hadoop classpath 依赖, 或者在 _common 目录下增加 Hadoop 相关配置文件 (core-site.xml、hdfs-site.xml、mapred-site.xml、yarn-site.xml)。

6. 数据更新是怎么回事?

Druid 可以通过对数据重新摄入生成新的 Segment, 一旦新的 Segment 成功生成, 新生成的 Segment 就将替代旧的 Segment。

7. 如何更新历史数据?

Druid 中历史数据的更新基于 Segment, 如 Schema 变化 (新增或减少字段)、Metric 聚合规则变化、Roll-up 聚合粒度改变等, 可以利用离线方式重算数据, 然后通过重跑 batch-ingestion 任务的方式覆盖指定时间范围内的数据。

8. 如何监控批量入 Druid 方式的任務?

以 Hadoop 导入数据方式为例:

(1) 首先离线 ETL 处理后把待写入 Druid 的数据放置在指定 HDFS 路径中。

(2) 然后通过 inputSpec 发送 HTTP POST 请求 (http://overlord_ip:port/druid/indexer/v1/task) 给 Overlord 节点并获取返回的 taskId。

(3) 最后定期利用 taskId 向 Overlord 节点发送 HTTP GET 请求 (http://overlord_ip:port/druid/indexer/v1/task/task_id/status) 检查任务状态。

9. 数据时间格式如何设置?

格式	描述	示例
auto	由 Druid 自动判断时间格式, 若含有非数字字符, 则按 iso 解析, 否则按 millis 解析	

续表

格式	描述	示例
iso	ISO 8601 标准	2016-09-01T12:00:00+08:00
posix	从 1970 年 1 月 1 日（UTC）开始经过的秒数	1472702400
millis	从 1970 年 1 月 1 日（UTC）开始经过的毫秒数	1472702400000

10. 启动 Tranquility-HTTP 服务失败？

当 Schema 中定义有使用扩展包特性时，需要在服务启动时指定加载扩展包。以使用 ThetaSketch 为例，启动服务时需要在 JVM 参数中指定加载对应的扩展包。示例参数如下。

```
-Ddruid.extensions.loadList=['druid-datasketches']  
-Ddruid.extensions.directory=dist/druid/extensions
```

A.2 查询

1. Druid 是内存数据库吗？

早期的 Druid 的确被设计为内存数据库，所有数据都在内存中。后来，由于应用需要支持更多的数据、内存大小提升有限和 SSD 技术的发展，Druid 采用了内存映射加使用配置管理的方式，就是将最需要被查询的数据放在内存中，超出部分和不常用的数据放在磁盘或 SSD 中。

2. 什么是 Druid 的深度存储？

Druid 依赖深度存储系统保存 Segment 文件。如果历史节点挂掉，Druid 则可以从深度存储系统中加载 Segment 恢复数据支持查询。深度存储系统可以使用 HDFS、Amazon S3 等。

3. Zookeeper 会不会因为单点失效影响 Druid？

首先，Zookeeper 可以部署为多节点模式，且通常非常可靠。其次，即使 Zookeeper 全部失败了，Druid 的查询部分也可正常对外服务，但是新写入的数据将无法进入系统，因为 Druid 需要通过 Zookeeper 管理 Segment。

4. Coordinator 会不会存在单点失效问题？

Coordinator 的运行采用 Master/Slave 模式，即每次只有一个 Master 工作，多个 Slave 节点处于预备状态中。Coordinator 和 Zookeeper 一样，如果全部失效后，将影响新 Segment 的

分配和管理,但对已有的查询没有影响。

5. 查询是否会经过 Coordinator?

Coordinator 只会参与管理 Segment 而不参与查询过程,因此查询不会经过 Coordinator 节点。

6. Druid 和 Elasticsearch 的比较?

Druid 通过数据预聚合的方式压缩数据并提升访问速度,但是原始数据经过 Roll-up 后将丢失。Elasticsearch 保存原始数据并通过建立倒排索引的方式对文档索引查询,因此非常适合检索的场景。Druid 需要预先定义数据的 Schema, Elasticsearch 则 Schema free, 因此可以写入各种 JSON 数据。

7. 如何处理时区问题?

Druid 中的时间采用 ISO 8601 格式,表示为日期+时间+时区,例如“2016-01-18T23:41:00+0800”,且内部使用 UTC 时区作统一计算处理。如果想要使用本地时间,则需要在服务进程、查询和摄入的几个地方使用一致的本地时区。

- 服务进程时区设置,服务启动时需要增加 JVM opts 参数。

```
-Duser.timezone=UTC+0800 ...
```

- Druid 批量摄入任务时区设置。

```
"jobProperties" : {
  "mapreduce.map.java.opts": "-Duser.timezone=UTC+0800 ...",
  "mapreduce.reduce.java.opts": "-Duser.timezone=UTC+0800 ...",
  ...
}
```

- 查询时区设置。

```
"granularity": {"type": "period", "period": "P1D", "timeZone": "Asia/Shanghai"}
```

- 查询时间区段设置。

```
"intervals": ["2016-09-01T00:00:00+08:00/2016-09-02T00:00:00+08:00"]
```

该区段表示查询从当地时间的 9 月 1 日起到 9 月 2 日时间范围内的数据,其中 interval 的表示方式为前闭后开(即时间范围是大于等于 9 月 1 日且小于 9 月 2 日)。

8. 如何设置 Broker 节点的内存?

Broker 节点主要利用 JVM 堆内内存合并历史节点和实时节点的查询结果, 而堆外内存则供 group by 查询操作时处理线程使用。Broker 节点内存设置注意事项如下:

- JVM 参数-XX:MaxDirectMemorySize 指定最大的堆外内存大小。
- $\text{MaxDirectMemorySize} \geq (\text{processing.numThreads} + 1) \times \text{druid.processing.buffer.sizeBytes}$ 。

9. 如何设置 Historical 节点的内存?

Historical 节点利用堆外内存存储中间计算结果集, 并通过内存映射的方式装载 Segment。因此可以综合两方面的数据获得推荐内存配置。Segment 数据可用内存大小的计算公式如下:

$$\text{memory_for_segments} = \text{total_memory} - \text{heap} - \text{direct_memory} - \text{jvm_overhead}$$

Historical 节点内存设置注意事项如下:

- JVM 参数-XX:MaxDirectMemorySize 指定最大的堆外内存大小。
- $\text{MaxDirectMemorySize} \geq (\text{processing.numThreads} + 1) \times \text{druid.processing.buffer.sizeBytes}$ 。

10. Historical 节点配置的 maxSize 含义?

maxSize 用于设置一个节点的最大累计 Segment 大小 (单位是字节)。修改这个参数将影响内存和磁盘的使用比例。如果这个参数的值大于总的内存容量, 则很容易引起系统过于频繁换页, 导致查询过慢。

11. 查询结果返回为空的原因?

- 数据源不存在。
- 查询的字段不存在。
- 查询的数据时间范围不匹配。

12. 查询实时写入的数据时, 数据来源?

- 已完成切换 (Hand off) 的 Segment 数据将被加载至历史节点中供查询。
- 未完成切换 (Hand off) 的 Segment 数据由实时节点或中间管理者中的 Peon 提供查询。

13. 为何多次并行聚合查询性能会下降?

通常思路是从判断是否遇到 GC 瓶颈入手, 聚合查询 (比如 HyperUnique) 会用到更多的内存, 因此比较容易遇到内存使用上限的问题。

A.3 管理

1. Druid 支持的 Java 版本?

Druid 支持 Java 7, 但是大部分应用都已升级到 Java 8。如果运行的 JDK 还在使用 Java 7, 则建议升级到 Java 8。

2. Druid 的外部依赖有哪些?

Druid 的外部依赖很少, 主要是下面三个。

- 深度存储 (HDFS)
- 元数据管理数据库 (MySQL 或 PostgreSQL)
- 协调管理 (Zookeeper)

3. 在 Druid 中如何配置 Segment 的保留时间?

Druid 中已生成的 Segment 会永久存储在 Deep Storage 中, 但是可以利用 Coordinator 的管理页面 (http://coordinator_ip:port) 配置指定 DataSource 的 Segment 在本地服务器上的保留时长 (只有本地服务器上的 Segment 才能被查询)。配置参考如下:

- Forever Load, 永久保留。
- Interval Load, 只保留指定时间段的数据。
- Period Load, 只保留最近指定时长内的数据。

4. Druid 如何设置 DataSource 级别的访问授权?

当前没有比较直接的方式, 但是可以通过间接的方式进行访问权限控制。比如设置代理服务访问 Druid 集群, 把权限检查放到代理模块中。

5. 如何做到 Druid 服务升级不停止对外服务?

升级顺序如下:

- (1) 历史节点
- (2) 统治节点
- (3) 中间管理者节点
- (4) 查询节点

(5) 协调节点

因为涉及正在处理写入的任务,因此在中间管理者节点中有如下两种方式进行任务恢复。

- 设置 `druid.indexer.task.restoreTasksOnRestart=true`, 服务重启后可以恢复任务状态。
- 首先, 停止 MiddleManager 服务状态, 通过 HTTP POST 请求 `http://middlemanager_ip:port/druid/worker/v1/disable`, 该服务将不再接收新任务。然后, 确认该服务的所有任务已完成, 通过 HTTP GET 请求 `http://middlemanager_ip:port/druid/worker/v1/tasks` 查询。最后, 重启升级后的 MiddleManager 服务并开启任务接收, 通过 HTTP POST 请求 `http://middlemanager_ip:port/druid/worker/v1/enable`。

6. 如何控制 Druid 日志大小?

由于 Druid 采用 `log4j2` 来记录日志, 因此可以通过配置 `RollingFile appender` 来实现服务日志和任务日志的循环功能。

`log4j2` 配置: <http://logging.apache.org/log4j/2.x/manual/configuration.html>

7. 如何将 Druid 数据转移到另一个 Druid 中?

Druid 中有一个 `insert-segment-to-db` 工具, 它可以帮助将 Segment 插入到另一个 Druid 集群中。示例如下:

```
java
-Ddruid.metadata.storage.type=mysql
-Ddruid.metadata.storage.connector.connectURI=jdbc:mysql://localhost:3306/druid
-Ddruid.metadata.storage.connector.user=druid
-Ddruid.metadata.storage.connector.password=diurd
-Ddruid.extensions.loadList=["mysql-metadata-storage","druid-hdfs-storage"]
-Ddruid.storage.type=hdfs
-cp $DRUID_CLASSPATH
io.druid.cli.Main tools insert-segment-to-db --workingDir hdfs://host:port/druid/
storage/wikipedia --updateDescriptor true
```

A.4 应用

1. 如何做基数统计?

- 使用内置聚合器 HyperUnique，其基于 HyperLogLog 算法，支持并集操作，统计结果误差为 2% 左右。
- 使用扩展库 DataSketch，其基于 ThetaSketch 算法，支持交集、并集和差集运算，可配置最大去重精度。

2. 如何做留存分析?

比如求满足留存条件“第一周来网站浏览过商品 A 后，第二周又来该网站浏览购买了商品 A”的独立用户数，可以利用 **ThetaSketch** 聚合器分别求出满足上述两周条件的独立用户数，然后利用 **postAggregation** 对两个结果做交集运算，即为留存下来的独立用户数。

3. 如何求百分位的数据?

比如，在分析性能指标中求处理延迟分别小于 50%、90%、95% 的数据时，可以利用聚合器 Approximate Histogram 做统计估算。

附录 B

常用参数表

B.1 扩展

配置项	描述	默认值
druid.extensions. directory	存放 Druid 扩展相关文件的根目录，Druid 会从这个目录加载所有扩展	extensions，这是基于 Druid 工作目录的相对 路径
druid.extensions. hadoopDependenciesDir	存放 Hadoop 相关依赖的根目录，Druid 会根据 Maven 中 Hadoop 的 coordinate 加载相关依赖	hadoop-dependencies， 这是基于 Druid 工作目 录的相对路径
druid.extensions. hadoopContainer- DruidClasspath	Hadoop Indexing 启动 hadoop 任务时，这个配置项提供了一种方式可以让用户显式地设置 hadoop 任务的 classpath，默认这个路径是根据 Druid 的 classpath 和扩展的路径自动生成的，但在有些情况下，会引起 Hadoop 和 Druid 的依赖冲突，此时需要显式地配置该项	null
druid.extensions. loadList	配置要加载的扩展列表，它接收 JSON Array 格式。如果没有设置，则会从 “druid.extensions.directory” 目录下加载所有的扩展，如果配置为“”，则不加载任何扩展	null

续表

配置项	描述	默认值
druid.extensions. searchCurrent- ClassLoader	这是一个布尔型配置，它决定是否在 Druid 的 class- path 中自动搜索扩展，其默认值为 true，如果设置 为 false，则不会从 classpath 加载扩展	true

B.2 Zookeeper

配置项	描述	默认值
druid.zk.paths.base	druid 在 Zookeeper 中的根路径	/druid
druid.zk.service.host	Zookeeper 的 host 配置，这是必选项	none
druid.zk.service. sessionTimeoutMs	Zookeeper 的 Session 超时时间，单位为毫秒	30000
druid.zk.service. compress	布尔型标志，它决定写入 Znode 中的内容是 否采用压缩	true
druid.zk.service.acl	布尔型标志，是否在 Zookeeper 中设置 ACL 权 限	false
druid.zk.paths. propertiesPath	Zookeeper 属性路径	\${druid.zk.paths.base} /properties
druid.zk.paths. announcementsPath	Druid 节点 announcement 的路径，用于节点 宣告负责处理指定时段的数据	\${druid.zk.paths.base} /announcements
druid.zk.paths.live- SegmentsPath	存放 Druid 节点宣告负责的所有 Segment	\${druid.zk.paths.base} /segments
druid.zk.paths.load- QueuePath	历史节点的 loadQueue，用于加载或者卸载 Segment	\${druid.zk.paths.base} /loadQueue
druid.zk.paths.coordi- natorPath	用于 Coordinator 的选主操作	\${druid.zk.paths.base} /coordinator
druid.zk.paths.served- SegmentsPath	已废弃，功能同 liveSegmentsPath	\${druid.zk.paths.base} /servedSegments
druid.zk.paths.indexer. base	索引服务的根目录	\${druid.zk.paths.base} /indexer

续表

配置项	描述	默认值
druid.zk.paths.indexer.announcementsPath	保存每个 Middle Manager (中间管理者) 宣告负责的所有 Segment	\${druid.zk.paths.indexer.base}/announcements
druid.zk.paths.indexer.tasksPath	用于给 Middle Manager 分配任务	\${druid.zk.paths.indexer.base}/tasks
druid.zk.paths.indexer.leaderLatchPath	用于 Overlord 的选主	\${druid.zk.paths.indexer.base}/leaderLatchPath

B.3 Metric

配置项	描述	默认值
druid.monitoring.emissionPeriod	定时发送 Metric 的周期	PT1M
druid.monitoring.monitors	配置可选的收集器, 该配置项接收 JSON Array 格式	none (不使用任何收集器)
druid.emitter	使用哪种 emitter, 可选项有 noop、http、logging 和 composing。composing 配置可以同时使用多种 emitter	noop
druid.emitter.composing.emitters	composing 的 emitter 列表, 比如可以设置为 ["http","logging"]	[]
druid.emitter.logging.logLevel	参照 Log4j 的日志级别	info

B.4 Coordinator Node (协调节点)

配置项	描述	默认值
druid.host	当前节点的 host	InetAddress.getLocalHost().getCanonicalHostName()
druid.port	当前节点监听的端口	8081

续表

配置项	描述	默认值
druid.service	服务名, 作为 Metric 或 Alert 消息中的一个维度。建议每个节点设置一个不同的名字加以区别, 同时用于节点发现服务	druid/coordinator
druid.coordinator.period	Coordinator 定时执行的频率, Coordinator 定时根据内存中的集群状态对数据拓扑进行均衡, 同时轮询发现可用的 Segment, 根据均衡策略分配给历史节点去加载	PT1M
druid.coordinator.period.indexingPeriod	发送索引任务到索引服务的频率, 只有开启了合并 Segment 或者转换 Segment 的版本才有效	PT1800S (30 mins)
druid.coordinator.startDelay	Coordinator 初次执行延迟时留有时间缓冲, 以便能获取最全的数据	PT300S
druid.coordinator.merge.on	布尔型标志, 开启该选项, Coordinator 会尝试把多个小的 Segment 合并成一个最佳大小的 Segment	false
druid.coordinator.conversion.on	布尔型标志, 开启该选项, Coordinator 会将 Segment 的数据格式转换成最高版本	false
druid.coordinator.load.timeout	Coordinator 分配 Segment 给历史节点, 等待历史节点加载完成的超时时间	PT15M
druid.coordinator.kill.on	布尔型标志, 决定 Coordinator 是否提交 Kill 任务去删除无用的 Segment, 这里是硬删除, 会删除元信息存储以及深度存储中的 Segment, 删除以后不可恢复	false
druid.coordinator.kill.period	提供 Kill 任务的频率, 该值必须大于 druid.coordinator.period.indexingPeriod 的值, 而且只有开启了 druid.coordinator.kill.on 才有效	P1D
druid.coordinator.kill.durationToRetain	Segment 的保留期, Kill 任务不会删除保留期内的 Segment, 该值必须大于或等于 0。需要注意的是, 默认值无效	PT-1S
druid.coordinator.kill.maxSegments	每一次提交任务能删除的最多的 Segment 数量。该值必须大于 0, 只有 druid.coordinator.kill.on 开启以后才有效	0

续表

配置项	描述	默认值
druid.manager.config. pollDuration	轮询 config 表的频率	PT1M
druid.manager.segments. pollDuration	更新或发现可用 Segment 的频率	PT1M
druid.manager.rules. pollDuration	更新加载规则的频率	PT1M
druid.manager.rules. defaultTier	加载规则的默认层	_default

B.5 查询相关

配置项	描述	默认值
druid.server.http.num- Threads	处理 HTTP 请求的线程数量	10
druid.server.http.max- IdleTime	HTTP 链接的最大空闲时间	PT5M
druid.processing.buffer. sizeBytes	存放中间结果集的缓冲区大小，该缓冲区使用堆外内存。将该项设置为较大值时，会在中间结果集缓冲区聚合更多的数据，减少数据的传递。需要注意的是，Druid 会为每个处理线程开辟配置大小的缓冲区，设置为较大值时需要保障有足够大的堆外内存	1GB
druid.processing.buffer. poolCacheMaxCount	缓冲区池的容量的最大值	Integer.MAX_ VALUE
druid.processing.for- matString	命名当前查询线程的格式	processing-%s
druid.processing.num- Threads	处理查询的线程数量，一般为 processorsNum-1，留下一个线程用于加载 Segment	Number of pro- cessors - 1 (or 1)

续表

配置项	描述	默认值
druid.processing.columnCache.sizeBytes	缓存维度字典的最大值，任何大于 0 的值都表示开启缓存。开启维度字典查找的缓存以后，能显著提升维度上的 Aggregator 性能，例如 JavaScript Aggregator 或者 Cardinality Aggregator，但对于基数比较大的维度，有可能因命中率降低导致性能下降。开启此选项会带来 GC 的负担	0(disable)
druid.processing.fifo	处理队列是否采用 FIFO 的公平方式	false
druid.query.groupBy.singleThreaded	是否使用单线程处理 Group By 查询	false
druid.query.groupBy.maxIntermediateRows	Group By 查询的中间结果集的大小	50000
druid.query.groupBy.maxResults	Group By 查询返回结果集的最大值，对于返回结果集较大的查询可以通过在查询上下文中设置超过该值	500000
druid.query.search.maxSearchLimit	Search 查询返回结果集的最大值	1000

B.6 Historical Node（历史节点）

配置项	描述	默认值
druid.server.maxSize	历史节点加载 Segment 的最大容量（字节数）。但这并不是严格规定的容量上线，它只是给 Coordinator 一个参照值，根据该值制定加载计划	0
druid.server.tier	当前节点所属的层的名字	_default_tier
druid.server.priority	在分层架构中，优先级决定了哪些节点能被查询使用。值越大优先级越高。默认值（没有优先级）适合没有交叉副本（层与层之间没有数据存储交叉）的分层架构	0
druid.segmentCache.locations	Segment 分配给历史节点以后，历史节点从深度存储中下载缓存在本地磁盘上。该配置是历史节点缓存分配给它的 Segment 的路径	none（不缓存）

续表

配置项	描述	默认值
druid.segmentCache.deleteOnRemove	布尔型标志，一旦当前节点不负责某个 Segment 以后，将 Segment 文件从本地缓存中删除	true
druid.segmentCache.dropSegmentDelayMillis	删除 Segment 的延迟时间，单位为毫秒	30000（30 秒）
druid.segmentCache.infoDir	历史节点会缓存它负责的所有 Segment 的元信息，一旦节点重启以后，从缓存中加载那些 Segment 而不是等待 Coordinator 重新分配。该配置路径下保留所负责的 Segment 的元信息	\${first_location}/info_dir
druid.segmentCache.announceIntervalMillis	在从本地缓存加载 Segment 到内存中时，后台线程执行宣告负责 Segment 的定时时间间隔，如果将该值设置为 0，则会等待所有的 Segment 加载完成以后才会执行宣告负责	5000（5 秒）
druid.segmentCache.numLoadingThreads	从深度存储中下载 Segment 的并发线程数量	1
druid.segmentCache.numBootstrapThreads	从本地缓存中加载 Segment 的并发线程数量	1

B.7 Overlord 配置

配置项	描述	默认值
druid.indexer.runner.type	任务运行方式，可选项有“local”和“remote”	local
druid.indexer.storage.type	可选项有“local”和“remote”，该配置项标明任务信息存储在本地 JVM 内存中还是元数据存储中，若存储在元数据存储中，则当 Overlord 失败以后依然能够恢复	local
druid.indexer.storage.recentlyFinishedThreshold	获取最近完成的任务列表时，过滤掉当前时间和创建时间的差值超出设置时限的任务	PT24H
druid.indexer.queue.maxSize	同时活跃的任务数量	Integer.MAX_VALUE

续表

配置项	描述	默认值
druid.indexer.queue.startDelay	Overlord 进行队列管理之前的休眠时间	PT1M
druid.indexer.queue.restartDelay	在队列管理过程中发生异常以后重试的等待时间	PT130S
druid.indexer.queue.storage-SyncRate	同步 Overlord 的状态到底层持久化存储的频率	PT1M

以下是 Overlord 的运行模式为 “remote” 时的配置。

配置项	描述	默认值
druid.indexer.runner.taskAssignment-Timeout	分配任务给 Middle Manager 的超时时间，超时以后抛出 Error	PT5M
druid.indexer.runner.minWorkerVersion	分配任务给 Middle Manager 时，允许运行的 Middle Manager 最小版本	0
druid.indexer.runner.compressZnodes	布尔型标志，Overlord 是否允许 Middle Manager 对 Znode 的内容进行压缩	true
druid.indexer.runner.maxZnodeBytes	Zookeeper 中 Znode 的内容最大字节数	524288
druid.indexer.runner.taskCleanup-Timeout	Middle Manager 和 Zookeeper 链接断开以后，等待多长时间执行清理工作，将其附属的任务置为 fail	PT15M
druid.indexer.runner.taskShutdown-LinkTimeout	发起 shutdown 请求以后的等待超时时间	PT1M
druid.indexer.runner.pendingTasks-RunnerNumThreads	处理分配任务给 Middle Manager 的并发线程数量	1

启用 autoscale 时的附加配置如下。

配置项	描述	默认值
druid.indexer.autoscale.strategy	自动伸缩策略。可选 “ec2” 和 “noop”，ec2 适合部署在 AWS EC2 上，自动申请 EC2 实例	noop

续表

配置项	描述	默认值
druid.indexer.autoscale.doAutoscale	是否进行自动伸缩	false
druid.indexer.autoscale.provisionPeriod	定时检测是否需要添加新的 Middle Manager	PT1M
druid.indexer.autoscale.terminatePeriod	定时检测是否要删除 Worker	PT5M
druid.indexer.autoscale.workerIdleTimeout	Middle Manager 空闲（没有运行任务）的时间超过该值，可以考虑被终止	PT90M
druid.indexer.autoscale.maxScalingDuration	等待 Middle Manger 启动的最长时间，超过则放弃	PT15M
druid.indexer.autoscale.numEventsToTrack	跟踪自动伸缩相关事件（节点创建或销毁）的数量	10
druid.indexer.autoscale.pendingTaskTimeout	处于 pending 状态的任务等待多长时间以后尝试自动伸缩	PT30S
druid.indexer.autoscale.workerVersion	设置该值后，只影响自动伸缩创建的 Middle Manager 的版本，可以动态调整	null
druid.indexer.autoscale.workerPort	自动伸缩设置以后，创建的 Middle Manager 的监听端口号	8080

1. 动态配置

Overlord 可以通过调用 HTTP 接口的方式来动态调整 Worker 的行为。需要指出的是，以下所有提及的 Worker，均可认为是 Middle Manager。

通过 POST 方式将一个 JSON 对象提交到如下 URL：

http://<OVERLORD_IP>:<port>/druid/indexer/v1/worker

可以在 Http Header 中添加审计相关参数。

Http Header 参数	模式	默认值
X-Druid-Author	修改配置的人	" "
X-Druid-Comment	修改评语	" "

Worker 配置规范的样例如下：

```
{
  "selectStrategy": {
    "type": "fillCapacityWithAffinity",
    "affinityConfig": {
      "affinity": {
        "datasource1": ["host1:port", "host2:port"],
        "datasource2": ["host3:port"]
      }
    }
  },
  "autoScaler": {
    "type": "ec2",
    "minNumWorkers": 2,
    "maxNumWorkers": 12,
    "envConfig": {
      "availabilityZone": "us-east-1a",
      "nodeData": {
        "amiId": "${AMI}",
        "instanceType": "c3.8xlarge",
        "minInstances": 1,
        "maxInstances": 1,
        "securityGroupIds": ["${IDs}"],
        "keyName": "${KEY_NAME}"
      }
    },
    "userData": {
      "impl": "string",
      "data": "${SCRIPT_COMMAND}",
      "versionReplacementString": ":VERSION:",
      "version": null
    }
  }
}
```

提交 GET 请求到相同的 URL，会返回 Worker 的当前配置规范。上述配置只适用于亚马逊云的 EC2，你需要扩展代码，使自动伸缩特性运行在其他云环境中。

Worker 配置规范描述如下。

配置项	描述	默认值
selectStrategy	给 Middle Manager 分配任务的策略，可选项有 fillCapacity、fillCapacityWithAffinity、equalDistribution 和 javascript	fillCapacity
autoScaler	自动伸缩相关	仅当开启自动伸缩以后有效

想要查看 Worker 配置的审计历史，发送 GET 请求到如下 URL：

http://<OVERLORD_IP>:<port>/druid/indexer/v1/worker/history?interval=<interval>

interval 默认采用 Overlord 的运行时配置属性文件中的“druid.audit.manager.auditHistory-Millis”的值，默认值为 P1W。

想要查看最近 n 条审计历史，发送 GET 请求到如下 URL：

http://<druid/indexer/v1/worker/history?count=n

2. Worker 选择策略

(1) Fill Capacity

优先将一个 Worker 的容量分配满，才分配给下一个 Worker。分配时先将 Worker 按照当前使用容量排序，把任务分配给当前使用量最大的 Worker。

配置项	描述	默认值
type	fillCapacity	必须是 fillCapacity

(2) Fill Capacity With Affinity

在 Fill Capacity 的基础上，优先根据 DataSource 的亲缘性选择 Worker。先来了解一下具体配置。

配置项	描述	默认值
type	fillCapacityWithAffinity	必选项，而且必须是 fill-CapacityWithAffinity

续表		
配置项	描述	默认值
affinity	亲缘性，它记录 DataSource 和对应的 Middle Manager 列表的映射，是一个 Map 的 JSON 对象，格式见上述配置，Druid 不会执行 DNS 解析，所以“host”必须和 Middle Manager 配置的值一致	{}

Fill Capacity With Affinity 的选取规则如下：

- 根据 DataSource 在 affinity 中查找，如果存在，则根据其对应的 Middle Manager 列表中的“host”和“port”在集群中查找 Worker，转换成 affinityWorkers，如果 affinityWorkers 不为空，则在 affinityWorkers 上执行 Fill Capacity 策略选取 Worker。如果 affinity 中的“host”和“port”在集群中不存在，则会导致 affinityWorkers 为空。如果 affinityWorkers 为空或者 DataSource 不存在，则进入下一步。
- 在整个集群上使用 Fill Capacity 策略选取 Worker。

(3) Equal Distribution

均匀分布策略，把任务优先分配给最小使用容量的 Worker。

配置项	描述	默认值
type	equalDistribution	必选项，而且必须是 equalDistribution

(4) javascript

采用 JavaScript 函数实现自定义选取 Worker 的逻辑，函数需要传入三个参数。

- config，类型为 RemoteTaskRunnerConfig。
- zkWorkers，workerId 到可用 Worker 的 Map 结构。
- task，要分配执行的任务。

函数的返回值为 workerId，或 Null，说明没有可用的 Worker。

它适用于 Worker 选取逻辑简单但又经常变更的场景，通过 JavaScript 函数的方式快速、灵活地实现。如果逻辑复杂，且又不能简单地在 JavaScript 环境下测试，则最好使用扩展模块的方式，扩展当前的 Worker 选取策略。

配置项	描述	默认值
type	javascript	必选项，而且必须是 javascript
function	自定义的 JavaScript 函数	

(5) autoscaler

当前只有亚马逊云中的 EC2 能使用 autoscaler。

配置项	描述	默认值
minNumWorkers	集群中 Worker 数量的最小值	0
maxNumWorkers	集群中 Worker 数量的最大值	0
availabilityZone	EC2 环境配置	none
nodeData	描述如何启动 Node 的 JSON 对象	必选项，none

B.8 Middle Manager 配置

Middle Manager 可以把它的配置传递给其启动的 Peon 进程。Middle Manager 的配置如下。

配置项	描述	默认值
druid.indexer.runner. allowedPrefixes	传递给 Peon 使用的属性前缀白名单	"com.metamx", "druid", "io.druid", "user.timezone", "file.encoding"
druid.indexer.runner. compressZnodes	Middle Manager 对 Znode 是否使用压缩	true
druid.indexer.runner. classpath	Peon 的 classpath	System.getProperty("java. class.path")
druid.indexer.runner. javaCommand	Java 的执行命令	java
druid.indexer.runner. javaOpts	给 Peon 使用的 JVM 参数，已废弃，鼓励使用 javaOptsArray	" "

续表

配置项	描述	默认值
druid.indexer.runner.javaOptsArray	功能同上，采用数组的方式，可以正确处理参数中的空格和引号，类似于 ["-XX:OnOutOfMemoryError=kill -9 %p"]	[]
druid.indexer.runner.maxZnodeBytes	Znode 的最大字节数	524288
druid.indexer.runner.startPort	分配给 Peon 的起始端口，Middle Manager 从起始端口开始查找，如果未被占用则使用该端口；反之，加 1 继续查找	8100
druid.indexer.runner.separateIngestionEndpoint	摄入和查询隔离，使用不同的 Jetty 对象，分别绑定不同的端口，以及使用不同的线程池	false
druid.worker.ip	Worker 绑定的 IP 地址	localhost
druid.worker.version	Worker 的版本号	0
druid.worker.capacity	Middle Manager 能接收的任务数量的最大值	Number of available processors - 1

B.9 Peon 配置

虽然 Peon 继承了 Middle Manager 的配置，但是仍然可以使用如下前缀在 Middle Manager 中显式地配置 Peon 的参数。

druid.indexer.fork.property

Peon 的其他配置如下。

配置项	描述	默认值
druid.peon.mode	可选项有“local”和“remote”，设置为 local 时，意味着 Poen 作为单一的节点运行，不推荐	remote
druid.indexer.task.baseDir	Peon 使用的临时目录	System.getProperty("java.io.tmpdir")

续表

配置项	描述	默认值
druid.indexer.task.base-TaskDir	持久化索引的目录	\${druid.indexer.task.base-Dir}/persistent/tasks
druid.indexer.task.defaultHadoopCoordinates	HadoopIndexTask 依赖的 Hadoop 版本	org.apache.hadoop:hadoop-client:2.3.0
druid.indexer.task.defaultRowFlushBoundary	内存增量索引持久化的阈值，达到多少条以后持久化到磁盘中	75000
druid.indexer.task.directoryLockTimeout	等待僵尸 Peon 退出的超时时间，轮询获取文件锁，如果达到超时时间，且僵尸 Peon 没有释放文件锁，则放弃	PT10M
druid.indexer.task.gracefulShutdownTimeout	Middle Manager 优雅地退出时，等待任务结束的超时时间	PT5M
druid.indexer.task.hadoopWorkingPath	HadoopIndexTask 的临时目录	/tmp/druid-indexing
druid.indexer.task.restoreTasksOnRestart	Middle Manager 优雅地退出以后，重启时是否恢复执行任务	false
druid.indexer.server.max-ChatRequests	Chat Handler 的最大并发请求数量，设置为 0 意味着无限	0



张海雷

资深工程师。目前在优酷土豆广告技术团队负责Druid集群的维护。活跃在Druid中国用户组，Druid、Redis和Storm的开源项目代码贡献者。



高振源

热爱技术，爱智求真的后台开发和数据工程师。先后负责过广告DSP产品、QQ公众号精准投放平台、数据分析产品等研发工作。目前在腾讯SNG企业产品部，负责企点产品的数据平台工作。



许哲

腾讯后台开发高级工程师，先后参与了公司企业产品消息服务后台、QQ公众号后台、QQ公众号精准投放平台等研发，目前在腾讯SNG企业产品部，负责腾讯企点的后台和数据平台开发工作。

专家力荐

正如许多广为应用的开源项目，Druid是为解决某个特定问题而诞生的。我希望通过这本书，您将更深入地了解Druid，并用它为您的组织创造价值。

杨仿今 Druid项目主要创始人，Imply公司联合创始人，CEO

只有久经经验又乐于分享的大数据架构师，才有这样的功力，把实时大数据分析技术的原理与实践讲得这么系统与透彻。书中随处可见来自实践的真知灼见。阅读这本书，就如同由一位老司机带着开启的美妙旅程，一路轻松、兴奋、风景无限。

鲁肃 蚂蚁金服CTO

Druid作为一款优秀的实时大数据分析引擎，非常强大，与之伴随的是使用上的复杂性，因此理解Druid的架构和运行机制原理对于更好地使用Druid及定制化扩展显得尤为重要。

张雪峰 饿了么CTO

大数据实时多维度分析场景充满技术挑战，很高兴看到Druid最终完美地解决了我们客户的问题。大数据时代已经到来，Druid无疑是解决大数据多维度实时分析的最佳选择，本书则是一把打开该技术之门的钥匙。

徐琨 Testin云测总裁

Druid因其在快速查询、水平扩展、实时数据摄取和分析这三方面都有良好的支持，很好地满足了我们的需求。本书的几位作者都是Druid中国用户组中非常活跃的技术专家，他们在社区中的口碑是本书质量的保证，如果你对Druid感兴趣，这本书一定不能错过。

陈冠诚 Druid中国用户组发起人

TalkingData自2013年开始关注Druid项目，因为它的特性非常契合分析的业务场景，能解决海量数据的多维交叉分析问题。同时，为了增强其分析能力，我们也在把基于Bitmap的自研分析引擎Atom Cube融合到Druid中。拥抱开源社区的各种曲折，有苦有乐，不足道也，但是庆幸有许多热情的领路人，给予大家无私的帮助。相信这本书能够带大家领略Druid的魅力，让大家少走弯路，真正聚焦在对数据的探索上。

肖文峰 TalkingData CTO

Druid正在开创海量数据实时数据分析的时代，作为一家以技术创新驱动的公司，OneAPM幸运地在正确的时间选择了正确的技术构筑自己的后端处理平台，我希望OneAPM的经验能够给后来者以借鉴，本书作者之一麒璜是Druid技术在OneAPM落地生根的实践者，这本书一定能够给大家更多的启迪。

何晓阳 OneAPM创始人，董事长

开源软件在过去十年中蓬勃发展，特别是在大数据等新兴领域，开源软件逐渐在企业级应用中占有一席之地。我们很欣喜地看到Druid这样有中国元素的开源项目在这个过程中茁壮成长，被企业客户接受并在核心系统应用中部署。

刘隶放 Cloudera 大中华区技术总监

我用“大、全、细、时”四个字来总结大数据，传统数据库在这种数据特性下根本无法支撑。而Druid的出现，正好比较完美地满足了这四点，特别是对于维度变换不频繁的场景，非常适用。本书既讲解了Druid技术本身，也讲解了多维数据分析相关的知识，并对业内的分布式存储和查询系统都做了对比。想要系统掌握Druid技术，推荐阅读本书。

桑文锋 神策数据公司创始人，CEO

本书让读者深入了解Druid的架构设计、设计理念、安装配置、集群管理和监控，书中还介绍了一些高级特性和核心源码的导读，最后深入分析了Druid的最佳实践。本书采用由浅入深、循序渐进的方式介绍Druid，是一本非常难得的OLAP的实时分析系统经典书籍。

卢亿雷 AdMaster（精硕科技）技术副总裁

Druid是一套非常棒的大数据软件，本书也是一本非常棒的Druid课本。本书出色之处在于，不仅对Druid的架构以及细节有深入的阐述，而且有非常详尽的代码例子（code lab），甚至有一章专门介绍怎么安装和配置，非常适合工程师一边学习，一边上机实践。

王晔 AdHoc吆喝科技创始人，CEO

向在大数据行业从事多年的架构师、正在如火如荼地开展大数据相关工作的工程师，以及正在准备步入大数据行业的新手推荐这本书。

Sherman Tong 微软中国研发中心 高级研发总监

（排名不分先后）



博文视点Broadview



@博文视点Broadview



策划编辑：符隆美
责任编辑：葛娜玲
封面设计：李玲

欢迎投稿
邮箱：fulm@phei.com.cn
微信：429753563

上架建议：计算机/大数据

ISBN 978-7-121-30623-5



定价：79.00元

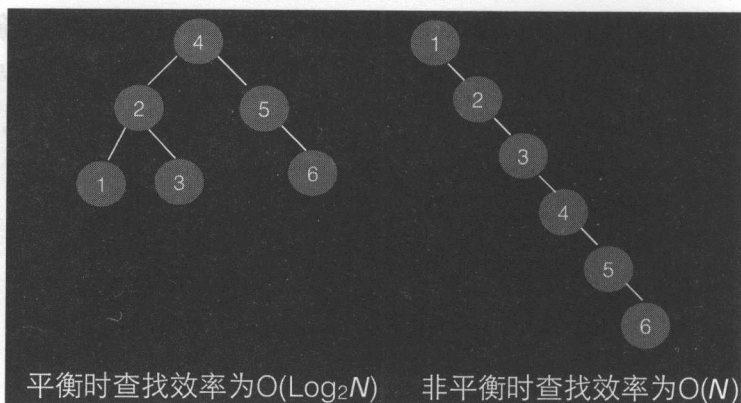


图 3-3 二叉查找树的查找效率

况且，主存从磁盘读数据一般以页为单位，因此每次访问磁盘都会读取多个扇区的数据（比如 4KB 大小的数据），远大于单个二叉树节点的值（字节级别），这也是造成二叉树相对索引树效率低下的原因。正因如此，人们就想到了通过增加每个树节点的度来提高访问效率，而 B+ 树（B+-tree）便受到了更多的关注。

2. B+ 树

在传统的关系型数据库里，B+ 树（B+-tree）及其衍生树是被用得比较多的索引树，如图 3-4 所示。

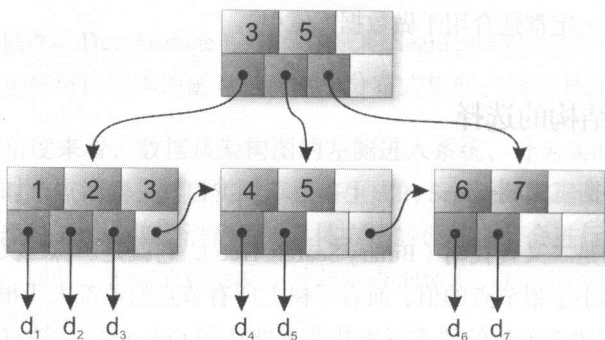


图 3-4 B+ 树

B+ 树的主要特点如下。

- 每个树节点只存放键值，不存放数值，而由叶子节点存放数值。这样会使树节点的度比较大，而树的高度就比较低，从而有利于提高查询效率。

- 叶子节点存放数值, 并按照值大小顺序排序, 且带指向相邻节点的指针, 以便高效地进行区间数据查询; 并且所有叶子节点与根节点的距离相同, 因此任何查询的效率都很相似。
- 与二叉树不同, B+ 树的数据更新操作不从根节点开始, 而从叶子节点开始, 并且在更新过程中树能以比较小的代价实现自平衡。

正是由于 B+ 树的上述优点, 它成了传统关系型数据库的宠儿。当然, 它也并非无懈可击, 它的主要缺点在于随着数据插入的不断发生, 叶子节点会慢慢分裂——这可能会导致逻辑上原本连续的数据实际上存放在不同的物理磁盘块位置上, 在做范围查询的时候会导致较高的磁盘 IO, 以致严重影响到性能。

3. 日志结构合并树

众所周知, 数据库的数据大多存储在磁盘上, 而无论是传统的机械硬盘 (Hard Disk Drive, HDD) 还是固态硬盘 (Solid State Drive, SSD), 对磁盘数据的顺序读写速度都远高于随机读写。磁盘顺序与随机访问吞吐对比如图 3-5 所示。

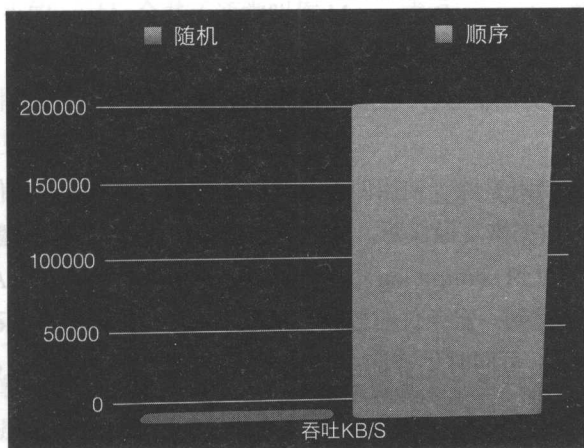


图 3-5 磁盘顺序与随机访问吞吐对比

然而, 基于 B+ 树的索引结构是违上述磁盘基本特点的——它会需要较多的磁盘随机读写, 于是, 1992 年, 名为日志结构 (Log-Structured) 的新型索引结构方法便应运而生。日志结构方法的主要思想是将磁盘看作一个大的日志, 每次都将在新的数据及其索引结构添加到日志的最末端, 以实现磁盘的顺序操作, 从而提高索引性能。不过, 日志结构方法也有明显的缺点, 随机读取数据时效率很低。1996 年, 一篇名为 *The log-structured merge-tree (LSM-tree)* 的论文创造性地提出了日志结构合并树 (Log-Structured Merge-Tree) 的概念, 该方法既吸收